THREAD AND PROCESS MANAGEMENT IN QNX NEUTRINO RTOS

TECHNICAL REPORT

Jakub Oleszczuk
Faculty of Transport, Electrical Engineering and Computer Science
University of Radom
Radom, Poland
118060@student.uthrad.pl

April 12, 2025

ABSTRACT

This paper presents the thread and process management mechanism of the QNX Neutrino real-time operating system. The aim is to highlight how QNX, as microkernel-based RTOS, handles concurrency and determinism in time-critical applications. The paper discusses the internal structure of threads and processes, their interaction, and real-time scheduling policies such as $SCHED\_FIFO$ and sporadic context of avoiding priority inversion and ensuring system reliability. The presented features make QNX Neutrino suitable for embedded and mission-critical systems.

Keywords QNX Neutrino · RTOS · Thread Management · Process Management

1 Introduction

RTOSs are designed to serve real-time applications that process data as it comes in, typically without any buffering delays. The QNX Neutrino RTOS [1] is a microkernel-based operating system that provides a high degree of concurrency and real-time performance. It is widely used in embedded systems, automotive applications, and other mission-critical environments. The QNX Neutrino RTOS is designed to be modular and scalable, allowing developers to customize the system to meet the specific needs of their applications.

1.1 RTOS vs General-Purpose OS

Real-time operating systems (RTOS) are designed to meet the timing constraints of real-time applications, while general-purpose operating systems (GPOS) are designed for a wide range of applications and do not have the same timing constraints. RTOSs are typically used in embedded systems, automotive applications, and other mission-critical environments where timing is critical. GPOSs are typically used in desktop and server environments where timing is not as critical.

1.2 QNX SDP 8.0

The QNX Software Development Platform (SDP) 8.0 is the latest version of the QNX Neutrino RTOS, which provides a comprehensive development environment for building real-time applications. The SDP 8.0 includes a wide range of tools and libraries for developing, debugging, and deploying applications on QNX Neutrino. The SDP 8.0 includes a new set of development tools, including the QNX Momentics IDE, which provides a powerful and flexible environment for developing real-time applications.

Advantages	Disadvantages
Fault isolation and recovery for high availability.	Requires more context switching, which can increase overhead.
Restart a failed system service dynamically without impacts to the kernel (no system reboot).	
Easy expansion—develop device drivers and OS extensions without a kernel guru and without recompiling.	
Easier to debug.	
Small footprint.	
Less code running in kernel space reduces attack surface and increases security.	

Table 1: Advantages and Disadvantages of a Microkernel RTOS (e.g. QNX Neutrino RTOS) [1]

1.3 QNX Neutrino

Although QNX SDP 7.0 was the classical distribution focused exclusively on Neutrino RTOS, the more recent SDP 8.0 (released in 2023) introduces enhanced development tools, still incorporates Neutrino's real-time capabilities with improved development tooling and broader system integration. The QNX Neutrino RTOS is a microkernel-based operating system that provides a high degree of concurrency and real-time performance. It is widely used in embedded systems, automotive applications, and other mission-critical environments.

1.4 QNX Momentics

The QNX Momentics IDE is a powerful development environment that provides a wide range of tools for developing, debugging, and deploying applications on QNX Neutrino. The IDE includes a code editor, a debugger, and a profiler, as well as tools for managing projects and building applications. The QNX Momentics IDE is designed to be easy to use and provides a wide range of features for developing real-time applications.

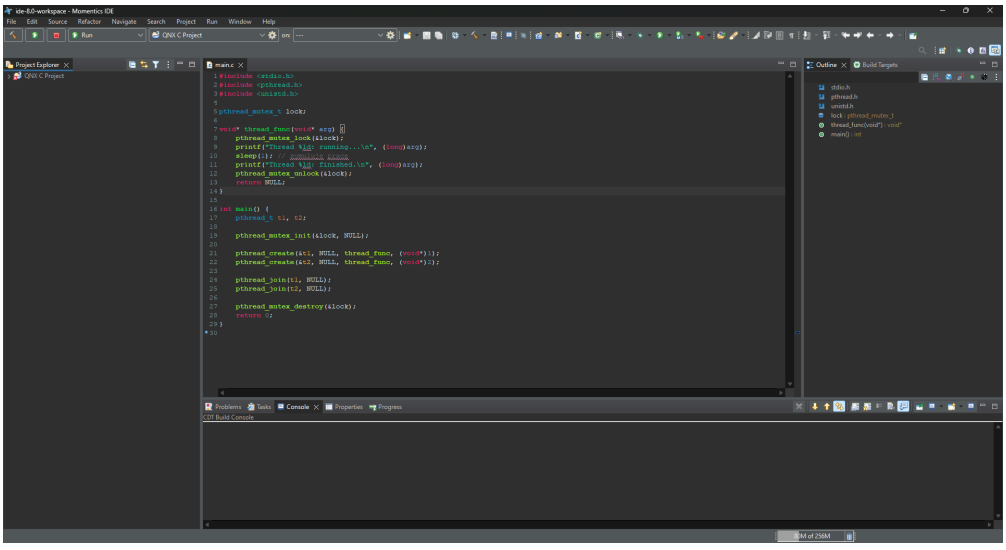

Figure 1: QNX Momentics IDE Window [2]

2 Monolithic vs Microkernel Architecture

For operating systems, the architecture can be broadly classified into two categories: monolithic and microkernel. Monolithic operating systems have a single large kernel that manages all system resources and services, while microkernel operating systems have a small kernel that provides only the most essential services, with other services running in user space. Monolithic kernels are typically larger and more complex than microkernels, as they include all the necessary services and drivers within the kernel itself. This can make monolithic kernels more difficult to maintain and debug, as any changes to the kernel can affect the entire system. Microkernels, on the other hand, are designed

to be modular and extensible, allowing developers to add or remove services as needed. This modularity can make microkernels easier to maintain and debug, as changes to one service do not affect the entire system. Monolithic kernels are typically faster than microkernels, as they do not require the overhead of inter-process communication (IPC) between user space and kernel space. However, microkernels can provide better fault isolation and recovery, as services running in user space can be restarted or replaced without affecting the kernel or other services. Monolithic kernels are typically used in general-purpose operating systems, such as Linux and Windows, while microkernels are used in real-time operating systems (RTOS), such as QNX Neutrino and VxWorks. The microkernel architecture is designed to be modular and extensible, allowing developers to add or remove services as needed.

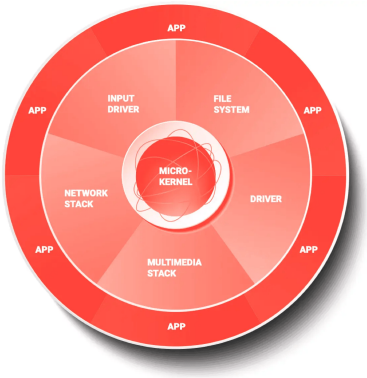

Figure 2: Microkernel OS architecture diagram [1]

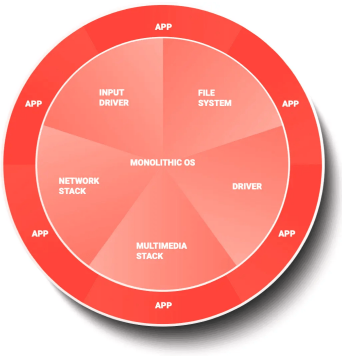

Figure 3: Monolithic OS architecture diagram [1]

3 POSIX Standard

The Portable Operating System Interface (POSIX) is a set of standards that define the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems. The POSIX standard is designed to provide a consistent and portable interface for developing applications that can run on different operating systems. The POSIX standard provides a set of APIs and standards for developing portable applications that can run on different operating systems.

3.1 QNX Neutrino RTOS and POSIX

The QNX Neutrino RTOS is designed to be compatible with the POSIX standard, which allows developers to use standard APIs and libraries when developing applications. The QNX Neutrino RTOS provides a wide range of tools and libraries for developing, debugging, and deploying applications, making it easy to develop real-time applications. The QNX Neutrino RTOS provides a POSIX-compliant API, which allows developers to use standard POSIX functions and libraries when developing applications. The QNX Neutrino RTOS also provides a set of extensions to the POSIX standard, which provide additional functionality and features for real-time applications. The QNX Neutrino RTOS provides a wide range of tools and libraries for developing, debugging, and deploying applications, making it easy to develop real-time applications.

3.2 QNX Neutrino RTOS vs Other RTOS (POSIX)

The QNX Neutrino RTOS is a real-time operating system that is designed to meet the timing constraints of real-time applications. It is widely used in embedded systems, automotive applications, and other mission-critical environments. The QNX Neutrino RTOS is designed to be modular and scalable, allowing developers to customize the system to meet the specific needs of their applications. The QNX Neutrino RTOS is based on the POSIX standard, which provides a set of APIs and standards for developing portable applications. The POSIX standard provides a set of APIs and standards for developing portable applications that can run on different operating systems. The QNX Neutrino RTOS is designed to be compatible with the POSIX standard, which allows developers to use standard APIs and libraries when developing applications. The QNX Neutrino RTOS provides a wide range of tools and libraries for developing, debugging, and deploying applications, making it easy to develop real-time applications.

4 Overview of QNX Neutrino Architecture

The QNX Neutrino RTOS is designed to be modular and scalable, allowing developers to customize the system to meet the specific needs of their applications. The architecture of QNX Neutrino is based on a microkernel design, which provides a small and efficient kernel that manages the most essential system services, while other services run in user space. The microkernel architecture of QNX Neutrino allows for better fault isolation and recovery, as services running in user space can be restarted or replaced without affecting the kernel or other services.

5 Thread and Process Management

RTOS systems are processing diffrently that general-purpose operating systems (GPOS). Threads and processes are the basic units of execution in both RTOS and GPOS. Threads are lightweight processes that share the same memory space and resources, while processes are independent units of execution that have their own memory space and resources. [3]

5.1 Thread Scheduling and Real-Time Constraints

In General-Purpose Operating Systems (GPOS), the scheduling of threads is typically based on a time-sharing model, where each thread is given a time slice to execute before being preempted by the scheduler. Most GPOS use preemptive multitasking:

- The OS decides which thread runs next.
- Threads are switched in and out based on priority, fairness, and time slices.

Example:

- Thread A is running.
- Its time slice ends $\rightarrow$ OS puts it on hold.
- Thread B is scheduled $\rightarrow$ it runs now.
- Context switch occurs (registers, stack, etc.).

In contrast, in Real-Time Operating Systems (RTOS), the scheduling of threads is typically based on a priority-based model, where each thread is assigned a priority level and the scheduler selects the highest-priority thread to execute next. This allows RTOSs to meet the timing constraints of real-time applications, where certain tasks must be completed within a specific time frame. [4]

5.2 Thread Priorities

Thread priorities are used to determine the order in which threads are scheduled for execution. In QNX Neutrino, thread priorities are represented by an integer value, with higher values indicating higher priority. The priority of a thread can be set when the thread is created or changed at any time during its execution. The QNX Neutrino RTOS supports a range of scheduling policies, including:

- **Round Robin** - Threads with the same priority are scheduled in a round-robin fashion, allowing each thread to execute for a fixed time slice before being preempted.
- **FIFO** - Threads are scheduled in a first-in-first-out order, with higher-priority threads preempting lower-priority threads.
- **Sporadic** - This policy is used for sporadic tasks that have a minimum inter-arrival time between executions.
- **Periodic** - This policy is used for periodic tasks that have a fixed period between executions.
- **Best Effort** - This policy is used for non-real-time tasks that do not have strict timing constraints.
- **Time-Slicing** - This policy is used for time-sharing tasks that have a fixed time slice for execution.
- **Priority Inheritance** - This policy is used to prevent priority inversion, where a lower-priority thread holds a resource needed by a higher-priority thread.

[5]

5.3 Thread Creation and Termination

In QNX Neutrino, threads are created using the `ThreadCreate()` function, which takes a function pointer and a set of attributes as arguments. The function creates a new thread and assigns it a unique thread ID. The thread can then be terminated using the `ThreadDestroy()` function, which allows the thread to clean up its resources before exiting. In GPOS, threads are typically created using the `pthread_create()` function, which takes a function pointer and a set of attributes as arguments. The function creates a new thread and assigns it a unique thread ID. The thread can then be terminated using the `pthread_exit()` function, which allows the thread to clean up its resources before exiting.[6]

5.4 Thread Synchronization Mechanisms

RTOS and GPOS systems use different mechanisms for thread synchronization. In GPOS, thread synchronization is typically achieved using mutexes, semaphores, and condition variables. In RTOS, thread synchronization is typically achieved using message queues, event flags, and other real-time synchronization mechanisms.[7]

6 Uses of Real-Time Operating Systems

Real-time operating systems (RTOS) are used in a wide range of applications, including:

- Embedded systems - RTOSs are commonly used in embedded systems, such as microcontrollers and digital signal processors (DSPs), where real-time performance is critical.
- Automotive applications - RTOSs are used in automotive applications, such as engine control units (ECUs) and advanced driver assistance systems (ADAS), where real-time performance is critical for safety and reliability.
- Aerospace and defense - RTOSs are used in aerospace and defense applications, such as avionics systems and missile guidance systems, where real-time performance is critical for safety and reliability.
- Industrial automation - RTOSs are used in industrial automation applications, such as robotics and process control, where real-time performance is critical for safety and reliability.
- Telecommunications - RTOSs are used in telecommunications applications, such as base stations and network routers, where real-time performance is critical for data transmission and processing.

7 Other Real-Time Operating Systems

There are several other real-time operating systems (RTOS) available in the market, each with its own features and capabilities. Some of the most popular RTOSs include:

- FreeRTOS - A popular open-source RTOS that is widely used in embedded systems and IoT applications. [8]
- VxWorks - A commercial RTOS that is widely used in aerospace, defense, and industrial automation applications. [9]
- RTEMS - An open-source RTOS that is widely used in aerospace and defense applications. [10]

8 Future of QNX Neutrino RTOS

The future of QNX Neutrino RTOS looks promising, as it continues to evolve and adapt to the changing needs of real-time applications. The QNX Neutrino RTOS is expected to continue to be widely used in embedded systems, automotive applications, and other mission-critical environments.

9 Example Code for Thread Creation and Termination (QNX-Specific)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>

void* thread_function(void* arg) {
    printf("Thread %d is running...\n", (int)arg);
```

```
    sleep(1);
    printf("Thread %d is exiting...\n", (int)arg);
    return NULL;
}

int main() {
    int thread_id;
    for (int i = 0; i < 5; i++) {
        thread_id = ThreadCreate(NULL, thread_function, (void*)i);
        if (thread_id == -1) {
            perror("ThreadCreate failed");
            return EXIT_FAILURE;
        }
    }
    printf("All threads created successfully.\n");
    sleep(2); // Allow threads to finish
    return EXIT_SUCCESS;
}
```

The above code demonstrates how to create threads in QNX Neutrino using the `ThreadCreate()` function. Each thread executes the `thread_function()`, which prints a message, sleeps for 1 second, and then exits.

10 Example Code for Thread Synchronization (QNX-Specific)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>

int shared_data = 0;
pthread_mutex_t mutex;

void* thread_function(void* arg) {
    pthread_mutex_lock(&mutex);
    shared_data++;
    printf("Thread %d incremented shared data to %d\n", (int)arg, shared_data);
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    int thread_ids[5];
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < 5; i++) {
        thread_ids[i] = ThreadCreate(NULL, thread_function, (void*)i);
        if (thread_ids[i] == -1) {
            perror("ThreadCreate failed");
            return EXIT_FAILURE;
        }
    }

    sleep(2); // Allow threads to finish
    pthread_mutex_destroy(&mutex);
    printf("Final shared data value: %d\n", shared_data);
    return EXIT_SUCCESS;
}
```

The above code demonstrates how to use `ThreadCreate()` in QNX Neutrino along with `pthread_mutex_lock()` and `pthread_mutex_unlock()` for thread synchronization. Each thread increments the shared data variable while ensuring mutual exclusion using a mutex.

11 Conclusion

In conclusion, the QNX Neutrino RTOS provides a powerful and flexible framework for managing threads and processes in real-time applications. The microkernel architecture of QNX Neutrino allows for better fault isolation and recovery, as services running in user space can be restarted or replaced without affecting the kernel or other services. The thread and process management mechanisms of QNX Neutrino provide a high degree of concurrency and real-time performance, making it suitable for embedded systems, automotive applications, and other mission-critical environments. The QNX Neutrino RTOS provides a wide range of tools and libraries for developing, debugging, and deploying applications, making it easy to develop real-time applications. The QNX Momentics IDE provides a powerful and flexible environment for developing real-time applications, with a wide range of features for managing projects and building applications. The QNX Neutrino RTOS is a powerful and flexible operating system that provides a high degree of concurrency and real-time performance. The QNX Neutrino RTOS is widely used in embedded systems, automotive applications, and other mission-critical environments, making it a popular choice for developers of real-time applications.

References

[1] QNX Software Systems. What is an rtos?, 2024.

[2] QNX Software Systems. Qnx momentics ide, 2024.

[3] IEEE. Posix.1-2017, 1988.

[4] QNX Software Systems. Thread scheduling, 2024.

[5] QNX Software Systems. Thread priority, 2024.

[6] QNX Software Systems. Process and thread management, 2024.

[7] QNX Software Systems. Thread synchronization, 2024.

[8] FreeRTOS. Freertos, 2024.

[9] Wind River. Vxworks, 2024.

[10] RTEMS. Rtems, 2024.