Mixed-Criticality Computing with the Quest RTOS and Quest-V Partitioning Hypervisor

Richard West
Department of Computer Science, Boston University
Boston, Massachusetts, USA
richwest@bu.edu

Zhiyuan Ruan
Department of Computer Science, Boston University
Boston, Massachusetts, USA
zruan@bu.edu

Shriram Raja
Department of Computer Science, Boston University
Boston, Massachusetts, USA
shriramr@bu.edu

Rafiuddin Syed
Drako Motors
San Jose, California, USA
rafiuddin.syed@drakomotors.com

Abstract

This article provides an overview of the Quest real-time operating system (RTOS) and the Quest-V partitioning hypervisor. We summarize the system features and requirements to use the Quest software development kit (SDK). The SDK is capable of prototyping systems combining one or more instances of Quest with legacy services from a Yocto Linux guest. Secure shared memory channels provide the means to build secure and predictable systems of systems.

CCS Concepts

• **Computer systems organization** → **Real-time operating systems**; **Embedded and cyber-physical systems**; • **Software and its engineering** → **Rapid application development**; **Software prototyping**.

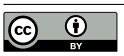
Keywords

Partitioning Hypervisor, Separation Kernel, Software Development Kit, Real-Time Operating Systems

ACM Reference Format:
Richard West, Zhiyuan Ruan, Shriram Raja, and Rafiuddin Syed. 2025. Mixed-Criticality Computing with the Quest RTOS and Quest-V Partitioning Hypervisor. In *International Conference on Embedded Software (EMSOFT '25), September 28-October 3, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3742874.3758338

1 Introduction

Most RTOSs today are incapable of single-handedly meeting the demands of highly complex embedded and cyber-physical systems, e.g.. in automotive, robotics, and industrial control environments. A natural solution is to combine a relatively small RTOS with a legacy system such as Linux, so the two are able to symbiotically benefit each other. The RTOS gains functionality that would take many person years to develop from scratch, while the legacy system gains predictable service guarantees.



EMSOFT '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1993-6/2025/09
https://doi.org/10.1145/3742874.3758338

The Quest-V partitioning hypervisor provides a means to co-host one or more instances of the Quest RTOS with Linux in a tightly-coupled system of systems. Together, these guest systems are able to support tasks that exchange information via secure shared memory communication channels. Quest-V differs from consolidating hypervisors by partitioning machine resources among guests into subsets of CPU cores, physical memory and I/O devices. Each guest operates like a distinct machine within a distributed system, or separation kernel, on the same physical hardware.

Quest-V's ability to host multiple instances of Quest is a form of *horizontal scaling*: applications requiring more hardware resources than available to one OS instance can be spread across guests.

The Quest SDK supports the rapid prototyping and development of applications specifically for the Quest RTOS, a Linux guest system, or a combination of guests. It enables testing and debugging without having to directly deploy a system onto hardware.

1.1 Quest Real-Time Operating System

Quest [1] is a relatively small RTOS, developed at Boston University. It works on uni- and multi-core processors, and supports various operating modes depending on the hardware features. It can be configured as either a lightweight SMP system, having a single memory image running on multiple cores, or as a secure separation kernel, using the Quest-V ("V for Virtualization") hypervisor.

The system features a novel real-time scheduling framework, where all control flows (including those triggered by interrupts) are associated with threads mapped to priority-aware, bandwidth-preserving virtual CPUs (VCPUs). VCPUs, in turn, are assignable to specific processor cores. This enables Quest to support both spatial and temporal partitioning of resources.

Key features of Quest include dual-mode (user-kernel) separation, combined task and interrupt scheduling using VCPUs, SMP support on processors up to 256 cores, local APIC timing, semaphores and spinlocks, synchronous and asynchronous inter-process communication, a real-time USB 2/3.x stack, and serial debugging.

Quest is POSIX-compliant, having a small libc implementation based on Newlib. It supports fork/exec/exit/waitpid process management system calls, as well as a subset of the pthread specification. Novel APIs allow the creation, migration and management of threads and VCPUs. A *shmcomm* API supports the creation of communication channels using FIFOs and 4-slot asynchronous buffers between processes in the same or different OSs.

1.2 Quest-V Separation Kernel

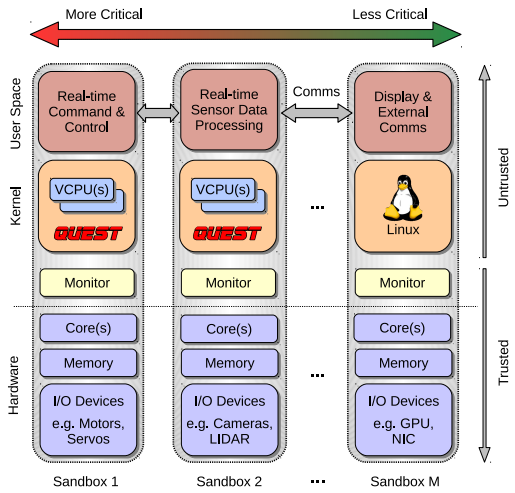


Figure 1: Example Quest-V Architecture Overview

Quest can be configured to run as a virtualized multikernel, taking direct advantage of hardware virtualization features to form a collection of separate kernels operating together as a distributed system on a chip. A Quest-V [2] multikernel is designed for high-confidence real-time systems, requiring operation in the presence of software faults. Quest-V also supports the hosting of Quest and Linux guests, connected via tightly-coupled, secure communication channels that support 4-slot (asynchronous) and ring-buffered (synchronous) information exchange.

Quest-V uses Intel VT-x hardware virtualization to isolate kernels and prevent local faults from affecting remote kernels. A virtual machine monitor for each kernel keeps track of extended page table mappings that control immutable memory access capabilities. An example Quest-V architecture is shown in Figure 1. In Quest-V, device interrupts are delivered directly to a kernel, rather than via a monitor that determines the destination. Apart from bootstrapping each kernel, handling faults and managing extended page tables, the monitors are mostly not needed. In special cases, they may be required to setup communication channels by manipulating extended page table mappings across sandboxes, or to assist in migrating address spaces, but otherwise, each sandbox kernel operates without requiring frequent guest/monitor (VM-Exit and Entry) transitions.

Quest-V differs from conventional virtual machine systems, which use a central monitor, or hypervisor, for scheduling and management of host resources among a set of guest kernels. Quest-V simplifies monitor functionality by avoiding runtime resource management. Consequently, the most secure protection domain (the hypervisor) has a small, trusted compute base, and is rarely accessed at runtime. This improves system efficiency and security.

2 Software Development Kit

The Quest SDK is able to bootstrap Quest and Yocto Linux on the same machine. Currently, the SDK works on KVM/QEMU using nested virtualization, with emulation of Intel Skylake 6th generation core-series x86 processors. The same package works directly on hardware up to 13th generation Raptor Lake Intel core-series x86 processors, as verified using a Supermicro X13SRN-H computer. The SDK comes with the following features:

- Resource partitioning of CPU cores, machine physical memory and I/O devices between Quest and Linux
- Predictable and secure shared memory communication between one or more instances of Quest and a Linux guest
- Coordinated power management across all guest OSs
- Quesh shell access from Linux to invoke services in Quest
- XFCE Desktop Window Manager environment
- The following programs are included in the SDK for development purposes: emacs, gcc, g++, automake, autoconf, binutils, make, gettext, pkgconfig, git, wget, openssh, gdb, usbutils.

Network connectivity is supported, for file transfer into and out of the SDK (e.g., using scp). Optional support exists for chromium-x11 and graphics acceleration using virtio-vga-gl.

2.1 Host Requirements

Listed below are the system requirements for running the SDK with support for both Quest and Yocto Linux:

- x86 CPU with hardware virtualization support
- At least 4 GB free RAM
- At least 8 GB free on-disk storage
- Linux kernel 5.x or newer with nested virtualization enabled
- QEMU/KVM
- SSH Server on the host if the user wishes to transfer file(s) between the SDK and the host machine

2.2 Quest Virtual CPUs

APIs exist to create and destroy VCPUs, get and set scheduling parameters, bind task threads to VPUs, and set their affinity to specific processor cores.

2.3 Shared-Memory Communication Channels

APIs exist for Quest and Linux to create and destroy both synchronous and asynchronous communication channels. They support the specification of marshall and unmarshall functions that are applied pre- and post-transfer of data across a channel, between processes on the same or different operating systems. Additional *tuned pipe* abstractions support real-time task pipeline construction and predictable shared memory inter-process communication.

3 Documentation

The SDK documentation explains how to configure Quest(-V), use its Docker toolchain to build the system, develop and debug applications. It includes information on the shmcomm, tuned pipe and VCPU APIs, system history and features.

References

[1] Matthew Danish, Ye Li, and Richard West. 2011. Virtual-CPU Scheduling in the Quest Operating System. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*. IEEE, USA, 169–179. doi:10.1109/RTAS.2011.24

[2] Richard West, Ye Li, Eric Missimer, and Matthew Danish. 2016. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Trans. Comput. Syst.* 34, 3, Article 8 (Jun 2016), 41 pages. doi:10.1145/2935748