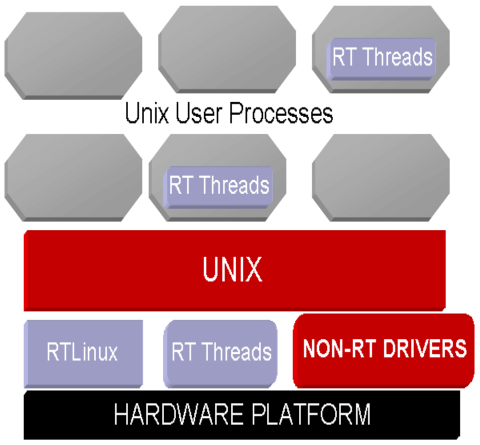
RTLinux/RTCore dual kernel real-time operating system

Victor Yodaiken, Cort Dougan, Michael Barabanov
{yodaiken,cort,baraban}@fsmlabs.com
FSMLabs Inc.

Abstract

OS modularity, real-time, communications, fault-tolerance, and security in the context of the dual kernel RTLinux and RTCore BSD operating systems.



1 Introduction

RTCore is a POSIX 1003.13 PE51 type real-time kernel, something that looks like a multi-threaded POSIX process with its own internal scheduler. RTCore can run a secondary operating system as a thread, using a small virtual machine to keep the secondary system from disabling interrupts. This is a peculiar model: a UNIX process with a UNIX operating system as a thread, but it provides a useful avenue to modularity. RTLinux is RTCore with Linux as the secondary kernel. RTCore BSD is, as one might guess, RTCore with BSD UNIX as the secondary kernel. Real-time applications run as real-time threads and signal handlers either within the address space of RTCore or within the address spaces of processes belonging to the secondary kernel. Real-time threads are scheduled by the RTCore scheduler without reference to the process scheduler in the secondary operating system. The secondary operating system is the idle thread for the real-time system. The virtual machine virtualizes the interrupt controller so the secondary kernel can preserve internal synchronization without interfering with real-time processing. Performance is adequate to allow standard PC and single board computers to replace DSPs in many applications. A one millisecond periodic thread running on a 1.2GHz AMD K7 PC shows worst case scheduling jitter of no more than 12 microseconds when the secondary kernel is under heavy load. The same example for a Compaq iPAQ PDA based on a 200MHz StrongArm shows worst case jitter of no more than 32 microseconds.

2 Modularity and architecture

The original intent of the RTCore split kernel design was to facilitate development of complex real-time applications needing both precise timing and use of services that are normally only found in a sophisticated and timing-imprecise operating systems. However, a kernel that can monitor and control the operation of a secondary operating system has a wide range of applications including networking, fault tolerance, and security. In order to preserve the benefits of the original design, we have had to emphasize a design rule that any activity that *can* go in the secondary system, *must* go in the secondary system. This design rule also appears to work well for recent extensions of the RTCore kernel to handle fault tolerance and security.

Experience with the RTCore kernels shows that there is an alternative to the traditional view of modularity in operating system design [1]. RTCore separates components algorithmically as well as by standard functional group. Low latency components can go in the real-time system, while higher latency components are left to the secondary operating system and its processes. An interrupt service handler may be implemented in either system, depending on the purpose of the device - and that purpose may change dynamically. A disk controller interrupt handler will usually be a component of the secondary operating system, but an A/D device handler will be in the real-time system. On the other hand, a network device may initially be under control of the secondary kernel, but may switch to real-time control. For example, we may let the secondary OS bring the system up, and then switch the network to real-time mode after it completes making connections. Different types of handlers have different implementation constraints even though they are both required to be “fast”. For example, even when called from interrupt context, the RTCore *semaphore post* operation forces an immediate context switch if it wakes a thread with higher priority than the current thread. The result is that if thread A is running and an interrupt causes higher priority thread B to be activated by a semaphore post operation, B will switch in before the interrupt handler returns. The purpose is to minimize the wake-up latency of higher priority threads, but this extremism in defense of low latency is probably out of place for a general purpose operating system.

[1]That is the system as a collection of functions or ”services”. But also note: ”we have defined the whole system as a society of sequential processes, progressing at undefined speed ratios” [Dij68].

The real-time kernel remains quite clearly separated from the secondary kernel, but there are overlapping or parallel functional systems. As an example, RTCore implements a small in-memory file system for pipes, shared memory, device I/O and networking. The real-time file system is designed with extensive use of simple lock-free algorithms[2] to reduce interrupt disable periods to minimum. Interaction with the file system of the secondary operating system is needed to allow non-real-time processes to read and write on fifos connecting to real-time threads and the use of non-real-time file systems. The interaction involves a asymmetric communication in which the real-time file system passes commands to the non-real-time file system and receives queued responses back. When real-time code creates a fifo, it can set a permission bit to request that the other end of the fifo be visible to processes running under the secondary operating system. In this case, the secondary operating system is requested to create the fifo inode in its persistent file system. When a process under the secondary operating system writes data to a shared fifo, the operation has standard POSIX semantics, but but is preemptible by the real-time side except during a short interval when buffer space is reserved. When real-time software executes a write command, the operation is non-blocking, non-preemptible by the secondary system, but preemptible on the real-time side.

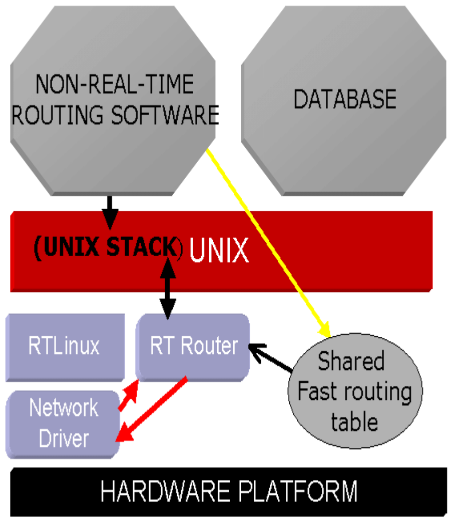
3 Communication and failover

Nearly every real-time operating system contains or supports some sort of network stack. Since our secondary kernels provide excellent network stacks, we use them for non-

[2]Following the approach of Massalin [MP91].

real-time operations[3] and add a simple zero-copy networking interface in the real-time system. When network devices are shared with the secondary operating system, the splitting method is used to provide the secondary operating system with a dummy network driver so that non-real-time packets can be passed through its network stacks.

As an example of how the split packet driver can be used, consider a fast router that runs under the real-time kernel and routes packets based on a small table in shared memory. The real-time driver picks up packets, looks for a match in the table, and forwards them back to the network (perhaps to a second device) on a match. When the match fails, the network driver drops the packet down to the secondary operating system stack and through that to a non-real-time routing program. That program may even consult a database system to determine what should be done with the packet and may then update the shared table (the methods of Degermark *et al* [DBCP97] are important here).



[3]For our purposes, UNIX and other secondary kernels are re-usable modules.

A fast failover method is also easily implemented in this design. Set a periodic thread to generate a ”keep alive” packet, say, every 5 milliseconds and to monitor some set of neighbors for such packets on the same schedule. A failure of a packet to arrive should trigger an alarm and cause a probe of the possibly failed site and perhaps a recovery action. Again, the complex components of the recovery can be delegated to processes running under the secondary operating system.

We are currently working on some applications to shaping of TCP/IP connections. The real-time kernel can periodically monitor the queues of network packets and the delays at TCP ports and can disable or discard packets belonging to less critical connections. While TCP itself does not provide for prioritization of traffic, the real-time network can transparently impose a prioritizing protocol on top of the TCP implementation of the secondary operating system, watch for timeouts, and even detect certain DOS type attacks.

As a final networking example, we anticipate being able to take advantage of real-time networking to efficiently use communications on clusters. Anecdotal experience indicates that sufficiently dense clusters suffer from sporadic significant delays when too many sites try to transmit at the same time. Precise timing control allows time domain multiplexing of bandwidth over communications media like Ethernet and even for dynamic adjustments to bandwidth allocations. It would be interesting to determine whether there are time allocation working sets on clusters.

4 Process Model and Memory Protection

The RTCore process model was originally single process. That is, the RTCore kernel appeared as a single process on a naked machine with applications as threads and signal handlers. The RTCore kernel and its applications all existed within supervisor memory so that we could avoid the costs of memory context switches on system calls and so that all of device space and the data structures of the secondary kernel were available directly. The system extended naturally to SMP systems by considering each processor to have a single RTCore process. While hardware speeds have increased, lowering the penalty of memory context switches, there is never any extra timing precision because additional timing precision makes new applications possible.

Two years ago, we were forced by application requirements to break from the original single shared address space to provide protected memory for some threads. Our original view was that the types of failures caught by memory protection hardware would invariably be fatal to a real-time application. If an application fails to stop a 20 ton hydraulic press at the right moment because of a stray pointer, the ability of the operating system to continue execution will not be comforting. However, as is usual with OS designers, we did not understand anything like the full range of applications. Customers using the system to do machine-in-the-loop simulation and manufacturing test reported that they used large, complex, and untrusted simulation codes from multiple suppliers and needed to be able to handle programming errors in these components. If the power supply simulation of a jet engine fails, it is still important to be able to gracefully turn off the fuel pump that is being used as part of the test. Memory protection was a first step towards such capability, but the need to keep latencies low and keep the real-time kernel simple was a complicating factor. The solution was to shift to a multi-process model in which we permit the creation of real-time threads within the memory space of processes belonging to the secondary operating system. These threads operate under the control of a real-time scheduler, but exist within the address space of a *host* process. When the real-time scheduler runs such a thread, it restores the host process memory map. Host processes must run on locked memory since paging is not compatible with execution of a real-time thread, but the host process can, for example, use the secondary OS I/O facilities to log data produced by its real-time threads.

Note that memory protection and a multi-process model does not overcome the essential ”single application” nature of a hard real-time operating system. Time is a shared critical resource and every real-time component must be able to affect the timing of every other one: otherwise we cannot say that the first component has any timing guarantees. Use of watchdogs and resource limits can mitigate the effects, but destroying or sidelining a thread means that its timing constraints have been violated. One of the utilities of the RTCore design is that there is a clear distinction between timing critical components and components that are not and the system makes sure that components of the second type cannot delay components of the first type. This decoupling facilitates the construction of sophisticated applications where back end processing is deferred to the non-real-time system.

5 Security

As a final note, security is becoming an issue for real-time systems in industrial and process control, communications, and medical instruments. The same properties that make RTCore useful for real-time, make it a valuable security kernel. A small real-time kernel that can even be placed in physically protected memory can be used to monitor the security of the secondary system, for example, by validating encrypted checksums on critical data structures. A watchdog implementation can then validate the integrity of the secondary kernel periodically and generate an encrypted packet for an external monitor. The external monitor may be anything from a computer on a local network or dedicated line to a special device on the local I/O bus, but it must be able to check for arrival of valid packets within specified times. Compromising such a watchdog would require compromising both the security of the secondary kernel and the watchdog in the real-time kernel and doing so without violating the externally visible timing constraints. Other variations of this type of monitoring are possible as well.

References

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.

[Dij68] Edsger W. Dijkstra. The structure of the “THE”-multiprogramming system. *Comm. ACM*, 11(5):341–346, 1968.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.