A Survey of Real-Time Operating Systems: Design, Implementation, and Comparison

Kelsey Anderson, Dan Gillen, Brian Hickson, Charles Wurster
University of California, San Diego

ABSTRACT

Real time operating systems (RTOSs) are a rich and growing field of computer science, both for research and industry. Many ideas and premises found in the study of non-real-time operating systems are applicable to RTOSs, but real-time systems take on a different focus: executing and completing tasks in a deterministic time frame. This paper is a survey of both research and commercial RTOSs, examining many of the decisions that went into their development and implementation.

INTRODUCTION

Early in their evolution real-time operating systems tended to be custom built and special purpose, bound to their target application or the hardware on which they would eventually execute. They tended to be relatively simplistic, embedded, centralized programs, where the programmer assumed the responsibility of resource management. Example applications were flight control and manufacturing systems. Further considerations for this era of RTOSs were that hardware was considerably more expensive than software, and was limited by size restraints and the need for low power consumption.

As technology has progressed over recent years, the development of RTOSs has grown in parallel with that of non-real-time O/Ss, or general-purpose operating systems (GPOSs). In particular, scalability factors in RTOSs have become pre-eminent with the ready availability of inexpensive hardware. Still, these real-time operating systems are dependent on many of the same principles and theories that drive non-real-time operating systems: efficient resource management, task communication and scheduling, load balancing, scalability, and synchronization. Unlike GPOSs, RTOSs must always keep focus on one primary goal: executing and completing tasks in a deterministic time frame. This focus is what defines RTOSs as being different from non-real-time systems. In fact, this difference often results in a tradeoff: performance enhancement vs. timing predictability.

Definition of Terms

A definition and discussion of some terms common to real-time operating systems follows:

Hard real-time: An activity or task is defined as hard real-time if it has a hard deadline for completion of action. A hard deadline is one that must always be met or the task is considered to have failed. Further, every hard real-time tasks in a system must meet its deadlines or else the system is considered to have failed.

Soft real-time: Real-time activities are said to be soft real-time if they are not defined as *hard real-time*. These activities usually have time constraints (deadlines) that can be missed without causing total system failure. Soft real-time deadlines may fall into more than the binary categories of hard real time deadlines (success/failure); example categories might be best, better, worse, or worst. In addition, soft real-time applications may contain code to recover from missing a deadline.

Microkernel: In comparison to the normal kernel of a GPOS, a microkernel performs only the most necessary functions. Examples of such functions are inter-process communication, scheduling, and interrupt fielding (not necessarily handling). Because many real-time systems

execute in a limited memory environment, it is helpful if the kernel is portable and small.

WCET: Short for Worst Case Execution Time. To guarantee timing requirements in real-time systems, the worst case execution time is always used for analysis.

POSIX: Portable Operating System Interface for Unix. POSIX is a set of standard operating systems interfaces that allow for the increased portability of applications. POSIX 1003.1b is reserved specifically for real-time operating system modifications, and supports interfaces for multiple threads (each with its own scheduler), reliable signal delivery, and timer resolution.

SURVEYED OPERATING SYSTEMS

Chimera II

The Chimera II real-time OS was developed at Carnegie Mellon University for sensor-based control applications like robotics and manufacturing process controllers. It was designed as a "local OS" to operate in a global/local OS framework. This type of computing environment is characterized by a host workstation running a global operating system like UNIX (may or may not also be a real-time OS), connected via a dedicated fast bus to one or several real-time processor units (RTPUs) which run a local real-time OS like Chimera II. In this computing environment, the global OS provides the programming environment (editor, compiler, linker, loader), while the local kernel provides other real-time functions such as task management, scheduling, interrupt handling, IPC, and memory management for the local RTPU. Chimera II only provides the local real-time kernel and relies on a UNIX-based host workstation to provide file system support and other services.

Maruti II

Maruti II is a second-generation system designed to support hard real-time applications in a reactive manner. A reactive system is capable of analyzing the resource requirements of new requests, verifying that such resources will be available given the time constraints, and then dynamically accepting or rejecting the new task for execution. This capability contrasts with simpler static and dynamic systems that can only accept new tasks and schedule them. The original Maruti system was designed as a set of real-time extensions to UNIX which provided a fault tolerant real-time development and run time environment based on the C programming language. Maruti II was implemented as a set of real-time extensions and modifications to Mach rather than UNIX because Mach offers more direct control over resources. It fulfills a broad series of design goals, including fault tolerance, distributed computing support, portability and extensibility. It provides both the programming environment and run-time kernel support for all types of applications, including hard/soft real-time applications as well as non-real-time applications. Its basic building block is referred to as an Elemental Unit (EU) and includes a programming environment for developers. This environment utilizes the Maruti Programming Language (MPL) and Maruti Configuration Language (MCL). The MPL and MCL are primitive extensions to C that describe EU configuration, timing, and resource information.

Spring OS

The Spring OS is based on the assumption that an RTOS must be constructed by looking at all the aspects of the system, starting from the hardware and continuing through to the programming language. Spring was developed at the University of Massachusetts as a distributed real time operating system. The hardware, operating system, and high level language (SDL) were all considered to ensure predictable results.

VxWorks

VxWorks is the most widely used real-time embedded operating. It was designed to be flexible and scalable. Various features of the OS are configurable and can be modified or removed for a given implementation. VxWorks supports the POSIX 1003.1b standard. The operating system is layered on top of a microkernel which

provides multitasking, scheduling, fast context switching, 256 priority levels, a variety of inter-task communications, interrupt handling, exception handling, dynamic memory management, and timing facilities. Layered on top of the microkernel are networking support, a local file system, and a flexible I/O system. The scalability of VxWorks allows it to run on a wide variety of platforms. All implementation decisions for this OS were also influenced by the requirement to sell copies, which is not an issue for research related systems.

RTLinux

RTLinux is a hard real-time extension of Linux that was developed at the New Mexico Institute of Mining and Technology. It allows for real-time applications to coexist with standard time-shared Linux applications. In RTLinux, the standard Linux kernel runs as a permanently pre-emptable task under a simple real-time kernel, allowing good average performance of time-shared applications while also supporting predictable and low-latency real-time behavior. RTLinux was designed to be flexible, with replaceable kernel modules for such things as scheduling and synchronization. RTLinux has been used successfully for many time-critical data collection and machine control applications.

RT-Mach

RT-Mach is a distributed real-time version of Mach that was developed at Carnegie Mellon to support on-demand applications, defense systems, vehicular control, and process control and manufacturing. It includes real-time thread management and synchronization, a time-driven scheduler, and memory resident objects. RT-Mach has also been integrated with a real-time tool set to aid development, containing tools for runtime monitoring, debugging, and visualization.

DESIGN ISSUES

The following sections investigate each of the above RTOSs with seven design issues in mind: hardware support, predictable memory management, predictable IPC and process synchronization, activity models (processes, threads, tasks), scheduling, and integration of RTOS and GPOS.

Hardware Support

A real-time system will not be predictable if the underlying hardware on which it depends is not also predictable. This requirement places strict guidelines on what kind of hardware support should be utilized in a real-time system. File access support and memory management are two good cases to consider. Disk-arm seek time and/or magnetic tape seek time due to a file read can cause an unpredictable time delay. Because of this, file access in real-time systems usually relies on RAM, limiting the amount of storage otherwise available to the system. Similarly, in memory management, page lookups that lead to cache misses can cause unpredictable time delays as well. As a result, RTOSs often access RAM directly. This unfortunately reduces the amount of memory and storage in the system.

Additionally, real-time systems are often designed for operating specific hardware or instruments with special requirements that may not normally be found in a general-purpose computing environment. Therefore, understanding the hardware needs and characteristics for real-time systems is critical.

Chimera II was designed to use standard hardware components. It supports VMEbus for fast communication between the host workstation and the RTPUs, RTPU-to-RTPU communication, and inter CPU (Motorola MC680x0) communication. The architecture of Chimera II would support other processor families, such as the Intel 80x86 or SPARC, with only a little additional work, though direct support was not included in Chimera II. It also provides interfaces to many different types of sensory and control devices such as actuators, switches, cameras, tactile sensors, and range finders. The programming environment for this hardware is based on the C language and library support, and so extends the general nature of the architecture.

Maruti II was designed to be platform independent so few assumptions were made regarding the underlying hardware. During its design and implementation, the designers used DECstations and Dell 433/MX PC compatible machines.

Spring was designed as a distributed system, consisting of multiple nodes. Each node is itself a distributed memory multiprocessor that is a collection of Motorola 68020-based MVME136A boards in a VME cage. Each of the boards has a 68851 MMU, and 4MB of local memory shared on the VMEbus. Each node has a SCRAMNet board that provides 2MB of reflective memory supported by a fiber-optic register insertion ring. SCRAMNet is used to interconnect these nodes. At each node there is a single board that is considered to be the System Processor, and it alone is responsible for the scheduling of tasks. The other node-local board are Application Processors, which serve to carry out the schedule created by the System Processor. Communication inside a node utilizes a round-robin mode of the bus to provide predictable access from one processor to the memory of another. The use of SCRAMNet is important because it provides predictability in inter-node communication. The use of other more common communications methods such as Ethernet or TDMA is either unpredictable or too slow.

The commercial VxWorks OS was designed to be as hardware independent as possible. An abbreviated list of supported targets includes PowerPC, ColdFire, MCORE, 80x86 and Pentium, ARM and StrongARM, MIPS, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20, and TriCore. VxWorks has extensions that allow for interactions with on-chip memory management units. In addition, VxWorks has extensions for multiprocessor systems of up to 20 processors.

RTLinux runs on the same hardware as Linux. No special hardware support is needed.

Predictable Memory Management

As noted previously, real-time systems are based not on the conventional notion of providing an excellent average case time of execution, but rather on providing a guaranteed Worst Case Execution Time (WCET). As will be seen, many of the memory management schemes of GPOSs are inadequate for real-time uses. The most common form of virtual memory management is based on a translation look-aside buffer that caches page mappings. In a real-time system, each memory lookup might require a TLB miss, causing a page fault and resulting in an abysmal WCET. Thus, virtual memory management presents some interesting challenges for any real-time operating system. Many systems, like Chimera and VxWorks, overcome the issues of effective virtual memory management by eliminating it altogether, and relying instead on physical memory only. Other operating system, like Spring and RT-Mach, try to come up with more creative solutions to virtual memory management.

Chimera II provides only a basic level of memory management limited by the memory available to the RTPUs and VMEbus. No virtual memory is supported since the designers knew of no way to implement virtual memory that would still provide the level of predictability required for a real-time system.

Spring OS uses a virtual memory system and attempts to keep the WCET low. To succeed it must consider misses in the TLB and page faults, both of which are catastrophic to WCETs. Spring OS avoids page faults by pre-allocating, at process creation time, physical pages for every used page in a program's address. To prevent TLB misses, the TLB is explicitly managed. On a context switch, the mapping for all the pages in a process's logical space are loaded into the TLB. This adds overhead to the WCET for a context switch but improves it for address translation, which occurs more frequently. On a context switch between user processes, two switches actually occur. The first switch is to the kernel to determine what to run next, and the second switch is from the kernel to the new user process.

The standard VxWorks kernel never performs paging because it imposes severe and unpredictable delays in execution time. An

optional extension to VxWorks is available that provides interactions with a hardware memory management unit (MMU). The extension does not guarantee real-time performance; in the documentation users are warned that use of an MMU may result in non-deterministic performance.

All RTLinux tasks execute in the single kernel address space with no virtual memory. Using a single address space was primarily done for performance reasons. Besides eliminating the overhead of protection level changes, it can make object access very efficient because objects can be referred to with names, which are automatically converted to addresses by the dynamic linker. It also allows for easy and efficient context switches, which are performed in software because hardware switches on the x86 were considered too slow when the system was designed. The software context switcher simply needs to push registers onto a stack and then change the stack pointer.

Predictable IPC and Process Synchronization

The constraint that all actions be predictable has major implications for inter-process communication. There must be a bound on the execution time of the IPC primitives, protocol processing, and message propagation. The use of FIFOs, queues, and other buffers in real-time communication introduces expensive WCETs when every buffer is full. Larger buffers only translate into a longer worst case execution time. If messages need to cross processor boundaries in addition to process boundaries then the introduction of a network provides even more difficulties. Some common network communications, such as Ethernet, must be ruled out because of their unpredictability. On the other hand TDMA, which guarantees delivery times, has incredibly poor performance in large systems. In most distributed real-time operating systems, specially designed communications paths have to be devised.

There are two levels of communication support in Chimera II. Local mechanisms provide communication between tasks, while global mechanisms provide communication between the host workstation and RTPUs, or between different RTPUs. In the local environment, tasks can access the address space of all other tasks, which provides local shared memory support for inter-task communication. Tasks can also communicate or synchronize through local semaphores or user signals. Both binary and counting semaphores are supported. Chimera II also supports priority message passing between tasks on the same or different RTPUs. Communication between the host workstation and RTPU, or between different RTPUs, is provided through the Express Mail protocol. Express Mail is the lowest level of communication used by the system level software. It is used to provide some processor transparency so that all other communication mechanisms are processor independent. For example, Express Mail is used to initialize message queues for inter-task message passing. In addition to Express Mail, a global shared memory service is provided by the VMEbus. Synchronization is provided through spin-locks, which use atomic test-and-set instructions to obtain locks but require expensive polling. Tasks running on different RTPUs synchronize by using remote semaphores. Like spin-locks, remote semaphores use test-and-set operations to obtain locks, but unlike spin-locks, a task will block instead of polling for a lock.

Maruti II supports both the message passing and shared memory models of communication. Messages are sent between EUs along a uni-directional communication channel that is transparent to both location and architecture. Messages are asynchronous with respect to the sender, though the receiver may block depending on the configuration. Messages are always dequeued in FIFO order. Notice that this mechanism must guarantee delivery by the OS, since there is no acknowledgement capability in a uni-directional communication channel. Two EUs are not permitted to have any circular channel paths; the MCL preprocessor cannot extract configuration information properly if there are cycles in the Elemental Unit Graphs (EUGs)

which define the flow of control and resources for a particular application. Maruti II supports shared memory using task abstraction. Tasks contain multiple threads that all share a common address space that may be used between threads or between EUs for sharing data. Maruti II also supports shared memory partitions as a means of sharing data between tasks. However, these partitions provide only a simple memory buffer between tasks and it remains the user's responsibility to ensure safe access to shared data within these partitions.

IPC in Spring is quite similar to the mechanisms in Mach. The system provides kernel protected objects called ports, which are used to send messages. Ports can be used to send or receive messages synchronously or asynchronously. Ports are type-based on the type of process that uses them (real-time or non real-time) and the type of communication (synchronous or asynchronous). Messages have a fixed size, always pass information by value, and can have requirements indicating when they must be delivered to a port. Timing for asynchronous messages is simple: the sender's time computation is the worst case of the recipient plus twice the time for communication. For synchronous communication the task is more difficult. It is unacceptable to have a real-time thread wait indefinitely for a response. Fortunately the Spring scheduler knows about both the task sending the message and the task receiving it, and can insure that the receiving task is ready to run. Inter-processor communication is done over the SCRAMnet, which provides replicated shared memory. Writes to the SCRAMnet end up in a FIFO, with an unacceptable worst case transmission time. To overcome this problem, Spring allocates particular sections of the shared memory for each processor. All processors can read from the entire shared memory.

In VxWorks, the simplest way to perform inter-process communication is through shared data structures. Since all tasks run in the same address space, communication is fairly straightforward. The OS provides for mutual exclusion by interrupt masks, task preemption, or semaphores. Semaphores can be used for mutual exclusion and for task synchronization. VxWorks also provides the IPC abstraction of message queues. There are two implementations of message queues: VxWorks proprietary and the POSIX standard. The main difference between the two is that POSIX has multiple priority levels. Because it is a commercial system, VxWorks provides a wide variety of additional ways to perform IPC. Pipes are provided as a way to perform IPC through the I/O system, and sockets are provided to allow for network IPC. On top of the socket layer, RPC is supported. Both UNIX and POSIX style signals are also supported. In addition, VxWorks provides an IPC extension package containing hooks for CORBA and DCOM.

In RTLinux, FIFO buffers are used for IPC both between the real-time processes and the Linux kernel, and between the Linux processes. The communication is always non-blocking on the real-time side, which eliminates many synchronization problems that have affected other real-time systems. These buffers are part of the kernel address space, and the Linux user processes see them as ordinary character devices.

The RT-Mach communication facility provides IPC by priority based message buffers. The message port has the ability to propagate priority from sender to receiver as well as a way to specify which task should receive the message if multiple receivers are possible. RT-Mach provides synchronization by use of mutual exclusion and a lock variable, as well as a highest-priority policy for determining which task will receive the lock. There is also a priority inheritance mechanism whereby a low priority task within a critical section can temporarily inherit the higher priority of the task that is trying to enter. This avoids the unbounded priority inversion problem, where a high priority task is forced to wait forever because a low priority task has been preempted while holding the desired lock. For added flexibility, a thread can be non-

preemptable, preemptable or restartable with respect to a particular lock.

Activity models (thread, process, task)

All operating systems typically achieve execution using one of two activity paradigms: messages or procedures. With messages, work is done as the messages traverse system. In a real-time message based system it is important that the message flow be deterministic. In the more common procedure-oriented system, work is accomplished by processes calling procedures. Nearly all the real-time operating systems are procedure oriented and are based on processes. Most use the standard model of a process as a single thread of execution with an associated address space.

Chimera II is task-based, where tasks map to threads or lightweight processes in traditional operating systems. General task management features like task creation, suspension, restart, preemption, and scheduling are built in to the kernel. Tasks can spawn other tasks; however, there is no hierarchical relationship between tasks. There is also very little inter-task security or protection. Chimera II was designed to be a single-user system operating and controlling sensor-based machinery, so very little security was designed into the system. Since a loose security policy between tasks can result in tasks corrupting other tasks, a global error handling mechanism has been provided to automatically detect bugs, memory corruption, and bad arguments in system calls. This error mechanism is provided in addition to the standard UNIX errno-based error handling scheme.

Maruti II provides a unique model for processes. The sequential set of EUs connected by uni-directional channels that form a path of data-and-control flow is known as a thread. All EUs share the processor state and execution stack and may send and receive messages from both inside and outside the thread. A job is a collection of these threads that cooperate with each other to provide some functionality and form a complete EUG. All threads within the job execute within the same time constraints, and scheduling occurs on the granularity of jobs. Tasks are collections of procedures, functions, and data structures defined by one or several jobs that share a common address space. Tasks in Maruti II map directly into programmed modules using the MPL and MCL language support. Processor allocation occurs on a task granularity to ensure that all data structures in memory are available to the processor during execution. This abstract model is supported fully in the MPL and MCL extensions in the programming environment and provides the basis for scheduling and processor allocation within Maruti II.

SpringOS uses processes to describe computations. Each process is made up of a single thread of control in an independent address space. Spring breaks each process up into tasks for scheduling and execution purposes. The resources that each task will use are known; therefore, the process can be scheduled in such a way that the tasks it needs to run will block as infrequently as possible. Processes are well defined using the System Description Language (SDL).

VxWorks organizes the execution of programs into tasks. Tasks have shared access to most system resources, and each task has some context. The context information associated with each thread includes the program counter, registers, a stack, I/O assignments, a delay timer, a timeslice timer, kernel control structures, signal handlers, and debugging and performance monitoring values. One important resource not associated with a task is an address space. All the code in VxWorks is executed in a single address space.

Scheduling

Scheduling provides one of the fundamental differences between an RTOS and a GPOS. GPOS schedulers generally strive to balance efficiency and fairness, achieving good overall performance without keeping any particular process waiting too long. In an RTOS, the scheduler is only concerned with one thing: scheduling a process such that its execution time

completes before its deadline has expired. This could potentially cause a non-real-time process without a deadline to wait forever.

Many scheduling approaches have been used in RTOSs to try to increase efficiency while always respecting the deadline constraints. The simplest is a static cyclic strategy that repeatedly executes a pre-computed list of tasks. It offers incredibly predictable execution, but a single good ordering is often difficult to construct, especially for a complicated system. The other fundamental static scheduling algorithm is Rate Monotonic (RM), which assigns fixed priorities based on the frequency of the task. Obviously, the static techniques' lack of support for dynamically changing periods and priorities might present a problem with certain systems. At the opposite extreme would be a dynamic priority scheduler that can increase efficiency, but sacrifices some predictability (especially in uncommon cases). An algorithm that can employ either dynamic or static strategies is Earliest Deadline First (EDF), using the deadline of a task as its priority.

There are several common scheduling algorithms supported by the Chimera II scheduler that allow both static and dynamic scheduling. Most common real-time systems provide RM static priority scheduling. The scheduler also supports EDF, minimum-laxity-first (MLF), and maximum-urgency-first (MUF) dynamic priority scheduling. The MUF scheduling algorithm was developed especially for Chimera II and provides support for dynamically reconfigurable systems (e.g., sensor-based systems). The scheduler is modular and can be replaced by any other user-defined scheduler.

Maruti II schedules both execution time and other resources dynamically through the use of a "calendar" data structure that captures the constraints of each EU with respect to time and other reserved resources. Calendars are built on the granularity of an EU, so scheduling can occur dynamically along that granularity as well. The designers have developed new techniques called temporal analysis and decomposition scheduling to allocate resources among EUs efficiently. Scheduling occurs using time division multiplexing schemes (TDMA) across multiple tasks and is enforced by the kernel dispatcher unit. The dispatcher loads applications, manages calendars, sets up communication channels, and performs exception handling during run time.

In Spring OS, there is no priority driven scheduling mechanism. Instead, the scheduling algorithm integrates CPU and resource scheduling to provide guarantees. Real-time tasks are given “precedence” in the system because they have time requirements. This ensures that they run within the time limit, though often results in non-real-time tasks being delayed. By understanding the timing and resource requirements of a process, it is possible to break processes into smaller non-blocking periods of execution called tasks. The tasks can be scheduled so that the processes are guaranteed to meet their deadlines. When an entity asks the scheduler to schedule a process, first it is broken down into tasks. If all the existing tasks in the system can be run in addition to the new tasks without violating the deadlines of the processes, then the process is scheduled; otherwise, the entity is informed that the process could not be scheduled. The scheduling algorithm also allows for groups of processes. Process groups are allowed to give deadlines for when the entire group needs to complete.

VxWorks provides preemptive priority based scheduling as well as round robin scheduling. Round robin scheduling attempts to share the CPU fairly among all ready tasks of the same priority. Preemptive priority based scheduling can be augmented with round robin scheduling. If both scheduling algorithms are turned on, priority based scheduling will select the highest priority task to run; if there are multiple tasks all with the highest priority, then round-robin is used among those tasks. VxWorks also provides a way to disable preemption. A task can indicate that is should not be interrupted and will run until it gives up the processor voluntarily or it is finished. The system also allows for priority inheritance.

RTLinux allows a user-defined scheduling policy in the form of kernel scheduling modules. This feature allows the system administrator to experiment with various schedulers, and to determine which module/algorithm best matches the needs of the system. Two scheduler modules are currently in wide use. The first is a priority-based preemptive scheduler that assigns each task a unique priority level and always executes the task with the highest priority. The second, which is intended for periodic tasks (where deadline equals period), gives shorter period tasks higher priority. Interrupt driven tasks can also be supported by creating an interrupt handler that awakens a particular task. Of course, the Linux kernel task is always given the lowest priority, so it only ever runs when the real-time tasks have nothing to do.

The original Mach scheduling mechanism used processor and processor set objects, where both a thread and a processor belong to a particular processor set. In this mechanism, each processor set has its own run queue, and threads are not allowed to change queues once they have been assigned. Mach's fixed priority preemptive scheduling was too limiting for real-time computing, so RT-Mach has an integrated time-driven scheduler which uses a capacity preservation scheme. It first gives available processor cycles to hard periodic and sporadic tasks, then assigns the remaining cycles to soft real-time tasks based on priority. The scheduler has a layered structure with the scheduling policy separated from the thread switching mechanisms within the kernel. This makes it easy for each processor set to have its own scheduling policy and to change the policy as needed. The three main policies that exist are RM, fixed priority, and round robin.

Integration of RTOS and GPOS

There have been many approaches to the integration of general purpose operating systems and real-time operating systems, each of which has different design goals and environmental constraints. One common design is to build a real-time kernel from scratch, linking it to other development environments only through dedicated tip lines to download programs and report back results of program execution. This type of design is typified in systems like VxWorks, pSOSystem, and other small real-time kernels. Another design is to provide real-time extensions and kernel modifications to a standard GPOS like UNIX or Mach. Examples of this category include RT-Mach and RT-Linux. A unique solution in the Chimera II system integrates a host GPOS system and multiple real-time systems in a single environment where the real-time kernels can communicate with and make system calls to the host as if it were just another real-time kernel. A group of designers at the University of North Carolina at Chapel Hill created a single system with two independent, co-resident operating systems, one a real-time kernel and the other one a GPOS, through the use of small "executives" that time-shared the hardware between the two kernels. Other solutions have been to develop a GPOS with user-modifiable scheduling, user-modifiable memory and process management modules, and device drivers embedded with small real-time kernels.

There are many reasons for wanting GPOS and RTOS integration. One example is the need for a comfortable development environment for developing real-time applications. Often the developer wants the features provided by a GPOS during development, including wide application support for compilers, editors, linkers, and other tools; yet the developer must also test and debug the application in a real-time environment. The level of integration required by the real-time application often affects the design of the system that results. Multimedia applications may derive great benefit by residing in an environment that tightly couples both GPOS and RTOS paradigms of computing, but instrument controllers like robotics micro-controllers may require very little integration to operate. This diversity of application requirements has directly led to many different approaches to integration of general purpose and real-time system functions which span a

continuum from the isolated real-time kernel to the full general kernel with specific real-time drivers or modules.

Chimera II uniquely integrates the host workstation environment with the RTPU running environment. The host workstation appears to all RTPUs as another RTPU and may therefore make use of the global shared memory, remote semaphores, or message passing in the real-time environment to communicate with real-time tasks. It also allows non-real-time applications to run seamlessly in the same environment as the real-time tasks. In addition, host procedure calls are provided in the Chimera kernel to access the host workstation services that are not provided locally, such as file system support. These services are provided transparently to the RTPUs by the host workstation.

Already existing extensions to the operating system can be used to allow RTOSs and GPOSs to coexist. The support for loosely coupled distributed system allows a real-time VxWorks system to interact with an NT system, for example.

The Linux OS kernel is simply a task that runs under the RTLinux kernel, so it naturally supports both real-time and non-real-time tasks.

CONCLUSION

All of the discussed RTOSs have been successfully used in real-time systems, to varying degrees of success. RTOSs like VxWorks were specifically designed and marketed for industry and thus push scalability, hardware support, and portability as main goals. RTLinux, while also used in industry, is more research-related and does not provide the same suite of support packages as VxWorks.

The development and design of all these systems have resulted in some unique and refreshing features. For example, Chimera II provides deadline failure handling for soft real-time applications, as well as supporting both static and dynamic scheduling algorithms. SpringOS implements its scheduling algorithm without the use of priorities. VxWorks is most notable for the number of packages for extensibility: support for more complicated memory management, multiprocessors, CORBA, DCOM, and Java are all available for purchase. Maruti II implements the concept of *scenarios*, special files that describe collections of jobs and their real-time constraints. Scenarios describe the mode of operation for the real-time system.

Due to the decreasing prices of hardware over the last decade coupled with increases in the speed of processors and networks, the design of real-time systems has come into a new era. No longer are these systems small and centralized, custom designed for one particular hardware or application. In fact standards such as POSIX are specifically designed to help in porting an application from one RTOS to another, regardless of the kernel or hardware.

The six surveyed real-time operating systems in this paper represent a very small cross-section of RTOSs available today, yet they implement and address many of the common trends existing throughout the entire field of real-time computing.

References

[1] Michael Barabanov, *A Linux-based Real-Time Operating System*. July, 1997.

[2] Bollella, Gregory, and Kevin Jeffay, "Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems," IEEE Real-Time Technology and Applications Symposium, Chicago, IL, May 1995, pp. 4-14.

[3] Humphrey, Wallace and Stankovic, "Kernel-level Threads for Dynamic, Hard Real-time Environments", Real-Time Systems Symposium, August 1995.

[4] E. Douglas Jensen, *A Real-Time Manifesto*. October, 1996.

[5] Niehaus, Nahum, Stankovic, ann Ramamritham, "Architecture and OS Support for Predictable Real-Time Systems," Spring internal document. University of Massachusetts, Amherst, July 1995.

[6] Saksena, Manas, James da Silva, and Asok Agrawala, "Design and Implementation of Maruti-II," provided as Chapter 4 of "Advances in Real-Time Systems" by Sang Son (ed.), 1995.

[7] Stewart, David B., Donald E. Schmitz, and Pradeep K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, No. 6, pp 1282-1295, Nov/Dec 1992.

[8] M. Teo, M., "Real-time Multicasting for Group Communication in the Spring Kernel", Spring internal document. University of Massachusetts, Amherst, July 1995.

[9] M. Teo, M., "A Preliminary Look at Spring and POSIX.4," Spring internal document. University of Massachusetts, Amherst, July 1995.

[10] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao "Real-Time Mach: Towards a Predictable Real-Time System," Proceedings of USENIX Mach Workshop. October, 1990.

[11] VxWorks Programmer's Guide version 5.4

[12] Victor Yodaiken, *The RTLinux Manifesto*. Department of Computer Science, New Mexico Institute of Technology. 1993.