Real Time Operating Systems

CS384

Design of Operating Systems

Submitted by: John P. Kraus
Submitted to: Dr. Sebern
Submitted on: April 25, 1998

Table of Contents

Introduction……...……....1-2

Real-Time Kernels
Policies……..….2
Rules…….……..3

Types of Real-Time Operating Systems…….…….3-5

Hard RTOS……..…4
Soft RTOS……..…….5

Common Kernel Services…….5-8

Software Design and Constraints……8-9

Conclusions……...……...9-11

Real-Time Operating Systems

Introduction

What is a real-time operating system? A real-time operating system (RTOS) has many advantages and differences over commonly thought of operating systems such as UNIX, Microsoft Window/95/NT, or Apple's MacOS. Any operating systems that is considered to be "real-time" often has to adhere to a strict set of policies and rules, with perhaps the most distinguishable feature being that a real-time operating system is deterministic.

For an operating system, being deterministic is a guarantee that a software system will execute within a specified time constraint. In general when a software process executes, there are inputs into the process, the process executes by a strict set of rules and finally generates an output, much like a simple control system as illustrated below:

Events of an RTOS are performed within a specified period of time because of hardware and software interactions and thus can accurately measure the number of clock cycles it will take. Therefore any process which is not predictable is non-deterministic and therefore also not real-time.

The time period is not the only thing that governs the RTOS, it is also based on a strict hierarchy of commands. "In order to derive the fastest response possible from real-time hardware, along with some degree of programming flexibility in software, many users

have developed their own real-time kernels (Puttre 2)." These kernels control the operations of the system.

In general, this paper will discuss four topics about real-time operating systems. The first is real-time kernels which details policies and rules which govern the development of an RTOS. The second section will identify two different types of real-time operating systems. Following that, common kernel services available in an RTOS will be touched upon. Lastly, software design issues and constraints will be discussed.

Real-Time Kernels

Real-time application software is much harder to develop than normal software. As previously mentioned, real-time kernels have to adhere to a strict set of policies and rules. To paraphrase Embedded System Products, a developer of RTOS components, if kernel conforms to the following policies and rules, your system design will be efficient.

<u>Suggested Policies:</u>

- A RTOS should be a multitasking design in order to maximize the CPU's efficiency. To achieve this, the kernels must know about the internal workings of the CPU and be fully integrated with the hardware target. Because of development costs of an RTOS, it should be designed to execute on a number of different hardware processors.
- The multitasking kernel should be driven in response to internal and external system events.
- The kernel should support a number of independent or interrelated tasks each having its own priority associated with its scheduling importance.
- The kernel's performance should be, to the greatest extent, deterministic. These timed benchmarks are often what differentiates one RTOS from its competitors.
- The kernel should be designed and written in such a way that it imposes minimal overhead to the application tasks and should have small RAM requirements. This is a very important factor in selecting an RTOS for one's system. Often manufactures allow kernels to be scaleable so that application designers can tailor it for their system.

Suggested Rules:

- Common rules of task scheduling applies. That is to say if a higher priority task become ready to execute, it preempts the lower priority task and becomes the current task. A Null task is always has the lowest priority and must never be blocked.
- The kernel must be interruptible but not reentrant.
- An interrupt service routine (ISR) must not issue kernel system calls except to signal another event or to terminate itself.

Because rules, such as ones stated above, constrain a real-time operating system, it makes it much harder to develop than normal software systems. In addition to finding defects, real-time programmers are faced with a multitude of problems. When faced with a problem, programmers often turn to tools provided to assist them. These tools can be from a simple background debugger, to a logic analyzer to an emulator. Each of these have a working knowledge of the kernel which make them an asset.

Real-time operating systems are not fully developed using software, the hardware is also designed with the intention of a real-time kernel to run on it. If the hardware is designed specifically for an RTOS to run on it, the designing and coding of the kernel, although difficult, becomes a bit more simplified. Because of the close working relationship between hardware and software, increased speeds and efficiency of the system result. Real-time software applications often execute on custom-designed hardware. This is a rapidly growing segment of the market, which drives the evolution of kernels and often requires them to be scaleable.

Types of Real-Time Operating Systems

There are two thoughts of mind when it comes to a real-time operating system. Those that have critical time constraints and those that have softer time constraints. This is what differentiates the two types of RTOS's discussed in this section. These two types are

the hard real-time operating system, and the soft real-time operating system. They both perform the same functions, but in different fashions.

Hard Real-Time Operating Systems

The hard real-time system is required to complete a critical task within a guaranteed amount of time. A process is submitted along with the time necessary to complete its operations. The scheduler then does one of two operations:

- The first would be the allowance of the process, guaranteeing that the process will be completed on time.
- The second would be the rejection of the process. This is known as resource reservation. This type of guarantee requires that the scheduler knows the exact length of time each type of process the operating-system to perform.

Therefore, each operation must be guaranteed to take a maximum amount of time. This guarantee is impossible in a system with a secondary storage device, or virtual memory. "Hard real-time systems are composed of special-purpose software running on hardware dedicated to their critical processes (Galvin 142)."

Examples of where a hard RTOS would be used is where safety critical systems are required. The NASA space shuttles would qualify here. The shuttle systems are often thought to be the most complicated and critical systems designed today. On top of critical task requirements, there are interactions with literally thousands of devices and are backed up with redundant systems. Similarly FAA imposes a strict set of guidelines when it comes to software development for today's flight systems, as well does the FDA with medical equipment.

Soft Real-Time Operating Systems

Soft real-time operating systems are less restrictive by nature. A soft real-time system requires that the critical processes receive priority over less critical ones. This may

appear nice to start with, but it causes an unfair allocation of resources, and may result in log delays and even starvation for processes if used or designed incorrectly.

The implementing of soft real-time applications requires careful design of the scheduler and other related aspects of the operating system. System interactions must be known as well as the system must have priority scheduling giving the highest priority to real-time processes. In addition the dispatch latency must be small. The faster a process can start execute, sooner it is runable.

Examples of applications where a soft RTOS may be used are multimedia devices, graphic devices or most commonly found appliances in our homes. Most of the software systems that have a real-time kernel fall into this category from a simple remote control, blender or toaster, to a teletype phone, a personal digital assistant or even mobile phone or pager.

There are the two types of real-time operating systems, hard and soft. From this point on, the general concept of a RTOS will be discussed and not each individual type because the same concepts and features apply to both equally.

Common Kernel Services

Almost all real-time operating systems on the market provide a common set of services with the kernel. It is easy to expound in great detail upon each of these services provided however, as noted previously, the purpose of this paper was to give a general overview and provide information on a RTOS's.

The kernel must be concerned with the management of systems resources like the CPU, memory, and time. All of the system resources are shared among competing processes and this must be handled in a delicate manor. The CPU has to be shared to

increase efficiency and execute processes faster. The sharing of memory resources is essential because it is a finite resource. As previously described in this paper, time management is critical to a real-time operating system.

In a multitasking environment, the kernel must provide an orderly means of control from one task to another and efficiently use the resources available. The kernel is required to keep track of needed resources and the state of execution for each task before switching to another task, commonly referred to as a context switch. The context switch should be performed in a timely manner, which is to say if it is not performed at a correct time, a system failure will probably occur. Each task has an associated priority according to its importance on the overall system design. When a task of a high priority needs to be executed, the task scheduler will take over.

There are three main types of task scheduling provided with most real-time kernels: round robin scheduling, time sliced scheduling and finally preemptive scheduling. Simply, round robin scheduling is the oldest of these three methods and provides a 'polling protocol' which grants control to a task. Each task is responsible for determining conditions for its execution and how far. It then yields control of the CPU to an equal priority task or become blocked. Most kernels have a mechanism to remove blocked tasks from the ready execution-state so that others may run. Time sliced scheduling is very similar to round robin with the exception that each task is given a predefined slice of time, called a quantum, in which to run. Once a time quantum expires, the kernel forces the task to yield control to another task. Finally, preemptive scheduling is the most commonly used and preferred method of task scheduling. It supports both round robin and time sliced scheduling. The system uses priorities along with event driven operation.

Most available real-time kernels provide the following list of services but is not limited to:

Static and dynamic task services: Static tasks are typically those tasks that are created and initialized on startup of a software application and they do not change. The are available throughout the running of the application. Whereas dynamic tasks can be created, changed or destroyed during the running of an application. Typically, dynamic tasks are used for a very specific short-term service.

Queues and lists: Often the functions provided for queues and lists are generic in nature and allow vast configuration or them. Queues can either be first in-first out (FIFO) or last in-first out (LIFO). Lists can be single linked lists or double linked lists.

Semaphores: A semaphore is a protective means for handling system resources among competing tasks.

Mailboxes: A mailbox provides an interface between a message sender and a receiver task. Sending and receiving of mail can be very similar to your own personal mailbox at home or an email account. Only you have access to your mailbox, but many people can send to you. The same is with tasks, a single task owns a mailbox but can receive messages from many outside sources. In addition, messages can be sent to multiple mailboxes.

Synchronous and asynchronous transmission: A message sent synchronously will automatically wait for a message acknowledgment. The task will typically wait for a specified time-out period or generate an error. It is up to the application to handle the error in an appropriate manner. Asynchronous transmission sends a message but does not wait

on the action of the receiver. An asynchronous message does not need to have an acknowledgment.

Timers: The kernel provides timers based off of periodic interrupts from a system time base. The granularity of the timer ticks may be fixed or configured, however once set, it may not change. Typically general-purpose timers, time-outs, and elapsed time counting is provided. General-purpose timers synchronize a task with an event that takes place after a certain time passes. Time-outs allow some kernel services to be blocked by a task for a limited amount of time. This kind of timer is used with synchronous transmissions. Lastly, elapsed time counting measures the amount of time between two events.

Memory management: The kernel provides a means in which the RAM memory available may be partitioned into different partitions of any number of blocks and the size of blocks. The memory map is configured on the application start-up and does not change. Many kernels implement memory management using a singly linked list mechanism. When a task allocates a block, it removes it from the memory heap and associates it to the task. Conversely, when the task frees it, it adds it back into the heap. This mechanism is used to prevent memory fragmentation.

Software Design and Constraints

The simplest real-time system is one that is composed completely of hardware. This can be completed using simple control systems with feedback control as shown below:

"Simple controllers like this can be designed using hardware alone, but the more complex the controlling becomes, the more software is used for the design (Warner 1)." But what would hardware be without software? That is where operating systems come into play.

Designers incorporate real-time operating systems into their application to create, maintain and support the execution environment in which their application can execute. The RTOS handles all of their resource management cleanly, effectively and efficiently. As the complexity of a real-time application increases, a commercially available RTOS works to ones benefit. The designer can then spend their efforts on their application design and implementation without having to worry about how the RTOS does things. Only an understanding of the services and configuration is needed.

However if one was to design a custom real-time operating system, an intimate knowledge of operating systems is required. Always keep in mind that timing and resource management is the most critical parts of a real-time operating system. There are more policies and rules suggested for the design of an RTOS and a designer should consider each for their application needs.

Conclusions

A real-time operating system is deterministic, that is, it guarantees that a software system will execute in a specified time constraint. A software process executes

with certain inputs into the process, the process executes by the rules set before an operating system, and the output is generated. In order for the RTOS to operate at the fastest possible speeds, some users develop real-time kernels. These kernels control the operations of the system. These kernels adhere to a strict set of policies and rules, if the kernel conforms to the policies and rules, the design will be efficient for the task before it. Below is a summarization of the rules previously mentioned:

Suggested Policies:

- A RTOS should be a multitasking design.
- The multitasking kernel should be driven in response to internal and external system events.
- The kernel should support a number of independent tasks each having scheduling importance.
- The kernel's performance should be, to the greatest extent, deterministic.
- The kernel should be designed and written in such a way that it imposes minimal overhead to the application tasks and should have small RAM requirements.

Suggested Rules:

- Common rules of task scheduling applies, all based on process priority.
- The kernel must be interruptible but not reentrant.
- An ISR must not issue kernel system calls except to signal another event or to terminate itself.

These rules constrain a RTOS, thus making it harder to develop than normal software systems.

There are two types of execution for RTOS's, those with critical time constraints, and those with soft time constraints. These two types are the Hard RTOS and the Soft RTOS. The Hard RTOS is required to complete a critical task within a guaranteed amount of time. A process is submitted along with the time necessary to complete its operations. The scheduler then either executes the process, or denies it. The Soft RTOS is different however. Soft real-time operating systems are less restrictive, requiring that the critical

processes receive priority over less critical ones. This may appear nice to start with, but it causes an unfair allocation of resources, and may result in log delays and even starvation for processes if used or designed incorrectly.

Almost all RTOS's on the market provide a common set of services with the kernel. The kernel must be concerned with the management of systems resources like the CPU, memory, and time. Most available real-time kernels provide the following list of services but is not limited to:

- Static and dynamic task services
- Queues and lists
- Semaphores
- Mailboxes
- Synchronous and asynchronous transmission
- Timers
- Memory management

For a RTOS to operate properly, all these rules, and services should be followed to the strictest letter. If all the design rules and policies are implemented, and all the services are created in the RTOS, there should be no problem executing the function that the RTOS is designed to operate on.

Bibliography

Embedded Systems Product: RTXC, Real-Time Executive in C. Houston, 1994.

Galvin, Silberschatz. Operating System Concepts Fifth Edition. Masachusetts: Addison-Wesley, 1998.

Gonzalez, Michael, 1994 "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems." *IEEE Transactions on Software Engineering* Vol. 20 (1), p.13-14.

Kraus, Thomas. March, 1998. Interview and review for paper.

Levi, Shem-Tov and Agrawala, Ashok. Real Time System Design. New York: McGraw-Hill, 1990.

Puttre, Michael. Mechanical Engineering. New York, 1991.

Embedded Systems Programming CD-ROM Library, 1998-1994

Plauger, P.J., 1990 "Evaluating Real Time Operating Systems" Vol. 3 (2)

Ripps, David, 1998 "*Real Time Tasking"* Vol. 2 (8)

Sperry, Tyler, 1998 "*Real Time"* Vol. 2 (8)

Warner, Luke. Fuzzy Logic Applications in Real-Time Systems. 1997.