Real Time System Partition for Multithreading Applications

S. Jalaja, R. Sivaranjani,V. Tamil Mullai

Abstract— The time and space partitioning in real-time Avionics systems, has been widely embraced by the industry. We present the design of real-time file system (RTFS), a file system that complies with the emerging standard for a file system. RTFS provides real-time accesses to data stored on a variety of mass storage device. In addition to an interface complying with emerging standard, RTFS provides an application interface that complies with a subset of the POSIX standard. Task partitions communicate their file operation requests to RTFS via queuing ports; such ports are also used to deliver the responses from RTFS to the task partitions. The temporal behavior of RTFS is predictable and the response times for file operations are bounded. The design of RTFS handles a mix of hard and soft real-time File access requests. RTFS implements metadata journaling using on-board non-volatile memory devices to provide fast file updates and fast file system recovery on faults. Finally, RTFS includes facilities to support network-centric operations and a set of design and maintenance tools. This paper overviews the design of RTFS and describes the realization of the many unique features of RTFS.

Keywords- Real Time File System, Portable Operating System interface, Real Time Operating System(PSOS), Storage Area Network, Flash memory.

I. INTRODUCTION

Online mass storage facilities that provide real-time performance are becoming an essential component of emerging avionics systems. Such a trend is driven by the need to acquire and process large amounts of data and by the general need to decouple the producers and consumers of the information to be processed. This trend is also consistent with the broad goal to provide a service oriented architecture implicit with network-centric and autonomic operations across subsystem components. There is a further trend in the avionics industry to use partitioning in time and space, to economically architect these complex, software intensive avionics systems with coexisting safety and security critical modules with modules that are feature rich but are economically and technically infeasible to assure. These systems are expected to have a variety of storage access requirements ranging from guaranteed hard real-time access to flight and mission-critical data along with soft or non-real-time access to other types of stored data sets.

In this paper, we present the functional characteristics and the design overview of a file system (FS) that complies with the time and space partitioning with the additional goals to concurrently support the conflicting requirements of maximizing storage access bandwidth while guaranteeing both hard real-time accesses and fairly sharing the FS and the underlying storage media for soft and non-real-time accesses.

Manuscript received January 15, 2014.

S. Jalaja, VGN Laxshmi Nagar, Chennai, India
R. Sivaranjani, VGN Laxshmi Nagar, Chennai, India
V. Tamil Mullai, VGN Laxshmi Nagar, Chennai, India

One of the key characteristics for adoption in high assurance avionics systems is that the file system is a *user level partition*. Our design incorporates a variety of features for guaranteeing real-time responses for I/O requests that specify deadlines. These include the use of (a) deadline- driven scheduling that factors in the impact of disk seeking on the overall response times, (b) request preemption within the file system, (c) the pre- allocation of disk storage based on a specified file dimension, (d) the pre-allocation of disk buffers and (e) a non-volatile memory device, ubiquitous in most avionics platforms, for implementing asynchronous writes to the disk and for implementing a journal that can be used to quickly recover the file system on crashes. To implement the implicit time partitioning, we use a multi- threaded implementation that enforces several quotas on resources, requests and I/O bandwidth allocation for each partition.

To implement fair sharing of the file system for non- critical and non-hard real-time I/O requests, we use a two-level scheduling algorithm based on laxity. Several additional design features are incorporated to improve overall performance, predictability and system scalability. Users of the file system access its services through library functions that pass along the file operation requests and the parameters from a user partition to the file server. Two such libraries are provided: one that provides a subset of the POSIX FS standard and another that provides a set of primitives. The libraries also provides an optional set of extended APIs (application programming interface) that allow users to specify time bounds and other parameters to guarantee hard real-time access to files. An associated set of design and maintenance tools are also being developed to aid in "what-if" analysis, configuration specification and file system recovery. Planned features for our file system include the support for a variety of peripheral devices, support for Storage Area Networks (SANs) and remote file access.

II. BACKGROUND AND DESIGN GOALS

The avionics industry has embraced a new standard for the real-time operating systems used in implementing safety-critical and certifiable flight control systems. This dictates the use of a system of (computing) tasks that has the following characteristic.

(a) Tasks are grouped into partitions, where each partition comprises of a set of tasks.

(b) The set of partitions that have to be executed in a repetitive fashion is described as a schedule that specifies the order in which the various partitions need to execute

c) Each partition is allocated a fixed time quantum for its execution. As soon as the time quantum assigned to partition Pj is over, the executing task T is interrupted and the execution of tasks in the next partition Pj+1

*Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication*

is initiated. During the next time quantum for Pj, the execution of tasks in Pj resumes with the interrupted task T.

(d) Partitions interact with each other in a controlled fashion. Data exchanges between partitions are controlled and take place via queuing ports set up among the partitions. Each partition has its private address space. Thus, partitions are effectively isolated in space and interact with each other only in a limited manner using the queuing ports.

The key characteristics of an operating system support the isolation of task partitions in time and space. Isolation in time is achieved by permitting a task within a partition to execute only within the time quantum assigned to the partition that contains the task. Each partition is guaranteed its own time slot for execution to support predictable system operation. Isolation of task partitions in space is achieved by allocating memory regions to partitions and controlling the interaction among tasks within different partitions. Such isolation in time and space makes it possible to verify the timing and safety aspects of the avionics software, as required by the Federal Aviation Authority (FAA) for certifying safety-critical flight controls and avionics software.

As avionics system become more and more sophisticated, it becomes necessary for the task software to access data from bulk storage devices, such as disk drives, CD/DVD drives and Flash memory. Such bulk storage devices can be used for instance to store flight data logs in a continuous manner, to hold flight planning information, and to hold map data. The efficient storage and access of data on such bulk storage devices require the use of file system software, analogous to file systems that are found in traditional operating systems (OS). However, unlike traditional file systems, the file systems used in avionics, must provide the same Isolation and timing guarantees of data access requests as is provided by the OS.

The design goals of the FS prototype Presented in this paper provides a variety of additional features to extend the FS to emerging avionics needs, including fast error recovery, network-centric operations and accommodates a variety of mass storage devices. Specifically, the design goals of our FS are:

Predictable Behavior: This implies that file operations must have bounded completion time and thus permit the system designer to make safe assumptions about the temporal behavior of the FS.

Maintain Isolation in Space and Time: The isolation requirement is intrinsic to any A653 compatible system. This design goal implies that proper isolation be maintained in time and space across tasks *within* the FS partition itself, as these tasks perform file operations on behalf of tasks within other partitions.

Hiding and Minimizing Mass Storage Access Delays: Mass storage devices used within file systems have significantly higher access times compared to RAM. This is not only true for storage devices like hard disks and CD/DVD drives but also for any Flash memory device that may be used For bulk storage. To provide real-time performance (and, indirectly, to implement predictable behavior), this goal implies that as much of the delays involved for file operations be hidden as possible.

Fast Fault Recovery: Avionics file systems are relatively more susceptible to transient errors and faults compared to ground based systems. Such faults can generate data I/O errors as well as errors that impact the integrity of the file system. This requirement implies that the FS should have features to let it recover from such faults as rapidly as possible.

Supporting Distributed and Network- Centric Operations: The inevitable transition of avionics systems from a centralized architecture to a distributed, network-centric architecture implies that the mass storage facilities used by the FS may have to be remotely accessed.

BLOCK DIAGRAM

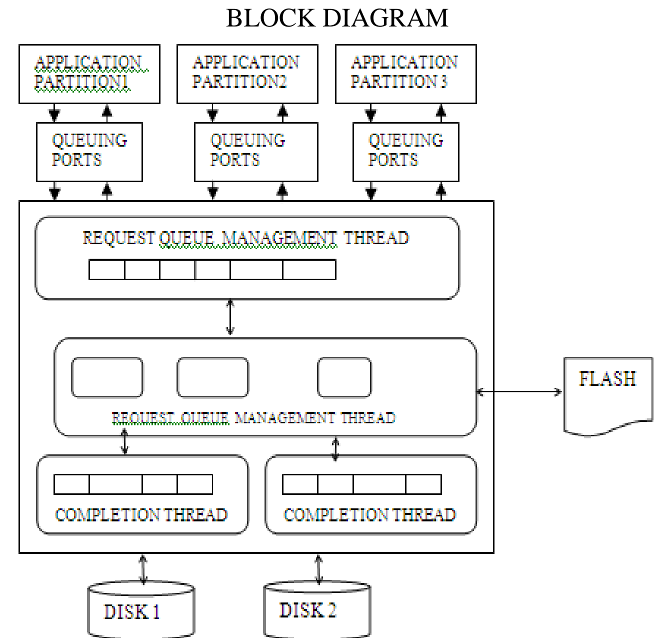


Figure depicts the main system-level components relevant to RTFS and some of the key internal components of RTFS. A single RTFS partition is shown in this figure.

III. OVERVIEW OF THE FILE SYSTEM

Our FS, hereafter called **RTFS** (Real-Time File System), is intended to operate in user space as a "partition" to support the isolation requirements of safety and security critical software as well as to enhance portability However, selectable options that tailor the RTFS as a "system partition" may be used to explore issues of performance in terms of latency and through put. The system designer can design a schedule where several task partitions are interspersed with one or more RTFS partitions within a major frame. Queuing ports (QPs) [1] are used to send file operation requests from a task partition to a specific RTFS partition within the schedule. The QPs are also used to deliver the results of such operations as well as status information to the task partitions.

An interactive tool that takes into account latency. And bandwidth requirements is provided to let the system designer determine the number of RTFS partitions to use and their placement with in the schedule.

RTFS provides an abstraction layer for managing persistent storage that is structured to support the determinism and response requirements of real-time avionics system applications. A typical use of RTFS will be to store flight data logs in a continuous manner and to store and retrieve in real- time map data to support enhanced

situational awareness, automated terrain avoidance, and dynamic route planning systems. Such facilities are critical to Free-Flight (FF) and Network Centric Operations (NCO). The broad design goals for RTFS are to concurrently support guaranteed real- time accesses while maximizing storage access bandwidth and fair sharing of the FS and the underlying storage media for non-real-time accesses. Additionally, RTFS must meet these goals by complying with the isolation requirements of time and space partitioning in the A653 standard.

Tasks within “application partitions” use library functions within an API (application programming interface) library to request various types of services from the RTFS. Such requests are submitted through queuing ports, as shown. QPs are also used to deliver responses from the RTFS to the application partitions. An application partition can use more than one set of QPs to a specific RTFS partition if needed. The RTFS itself is implemented by a set of threads. Our design does not exploit or assign any native priorities of such threads in the interest of portability. Instead, the required scheduling and orderings of the threads within the RTFS are explicitly implemented in the design. The RTFS implements the necessary synchronization and scheduling structures for the required thread scheduling. The main types of threads used within the RTFS are as follows:

Request Queue Management Thread (RQMT): This thread is responsible for dequeuing incoming file operation requests from the QPs and for copying arguments to the requests into an RTFS- internal high-level scheduling queue. The RQMT assigns a globally unique identifier to each request and performs any preemption that may be required to order requests within the RTFS-internal high-level scheduling queue. The RQMT is also responsible for delivering the responses to a request via the QPs.

Request Processing Threads (RPTs): Several instances of these “worker” threads are used to implement the processing needed to service a request. Each request is assigned to a RPT by the RQMT. For file operations that modify the file metadata (such as directory information, file allocation tables and soon), the RPT performing such updates log metadata transactions to a journal maintained within on-board non-volatile memory. This permits the file system to be quickly recovered on a crash, as metadata transactions remain saved on the non-volatile memory, allowing the recovery steps to restore the integrity of the file system. The RPTs also invoke device-specific low-level scheduling functions that mitigate the impact of seeking delays on mass storage devices such as disks and CD/DVD drives. RPTs also implement the support required for accessing mass storage devices remotely in network-centric environments.

Completion Threads (CTs): For each type of device used, an appropriate CT is provided to handle completions and low-level request scheduling for that device. The CT essentially pools the device status to determine when pending I/Os have completed. On the completion of an I/O request for the device, the CT for the device initiates the next operation for the device from the head of the low-level scheduling queue for the device.

IV. TIME AND SPACE PARTITION

Software in real-time embedded systems differs fundamentally from its desktop or Internet counterparts. Embedded computing is not simply computation on small devices. In most control applications, for example, embedded software engages the physical world. It reacts to physical and user-interaction events, performs computation on limited and competing resources, and produces results that further impact the environment . Of necessity, it acquires some properties of the physical world, most particularly, time. Despite the fact that both value and time affect the physical outputs of embedded systems, these two aspects are developed separately in algorithms may rely on. In most control applications, this run-time uncertainty is undesirable or even disastrous.

We believe that two steps can be taken to improve the de- sign process for embedded software and to bridge the gap between the functionality development and timing assurance:

- Rigorous software architectures that expose resource utilization and concurrent interactions among soft- ware components.
- Specification, compilation, and execution mechanisms that preserve timing properties throughout the software life cycle

A component-based software architecture can help compilers to determine typical embedded software design. The functionality is determined at design time with assumptions such as zero or a fixed nonzero run-time delay. The actual timing the logical dependencies and shared re- sources among components. By bringing the notion of time and concurrent interaction to the programming level, compilers properties are determined at run time by a real-time operating system (RTOS).

Typically, an RTOS offers as control of these timing properties one number for each task, a *priority.* Whether a piece of computation can be finished or brought to a quiescent state at a particular time is totally a dynamic phenomenon, and it depends largely on the hardware plat- form, when the inputs arrive, what other software is running at that time, and the relative priorities. These factors are usually out of the control of embedded system designers and may break the timing assumptions that the control algorithm and run-time systems can be developed to preserve both timing and functional properties at run time.

Recent innovations in real-time programming models such as port-based objects (PBOs) [1] and Giotto [2] are examples that take a time-triggered approach to scheduling software components and to preserving their timing properties. These purely time-triggered approaches, although explicitly con- trolling the timing of each component, require tasks to be periodic and do not handle irregularly spaced new information (or events) well. In this article, we introduce an event-triggered programming model, timed multitasking (TM), that also takes a time-centric approach to real-time programming but controls timing properties through deadlines and events rather than time triggers.

By doing so, each piece of information is processed exactly once, and the tasks can be a periodic.

This model takes advantage of actor-oriented software architecture [3] and embraces timing properties at design time, so that designers can specify when the computational results are produced to the physical world or to other actors. The specification is then compiled into stylized real-time tasks, and a run-time system further ensures the function and timing determinism during execution. As long as there are sufficient resources, the computation will al- ways produce predictable values at a predictable time.

V. DYNAMIC PARTITIONING OVERVIEW (PROPOSED TECHNIQUE)

A hardware partitionable server can be configured into one or more isolated hardware partitions. Each hardware partition in the server is assigned its own processors, memory, and I/O host bridges that are independent from all other hardware partitions in the server. Each hardware partition runs an independent instance of the operating system.

A hardware partition consists of one or more partition units. A partition unit is the smallest unit of hardware that can be assigned to a particular hardware partition. A partition unit can be an individual processor, memory module, or I/O Host Bridge, or it can be a hardware module or board that contains a combination of these components. Today's hardware-partitionable servers typically have multiple hardware components in each partition unit. For example, a single partition unit could consist of four processors, a memory module, and some I/O. In this situation, all of the hardware components in the partition unit must be added or replaced in a hardware partition as a single unit. As the number of processor cores per physical processor increases, the number of components per partition unit is likely to come down. However, with the memory controllers being implemented internal to the processors, processors and memory are likely to continue to be included in a single partition unit.

On a statically partitionable server, the configuration of partition units that are assigned to each hardware partition cannot be changed while the system is running. To change the configuration, the system must be powered down and restarted. Windows Server 2000 and later versions of Windows Server support statically partitionable servers.

On a dynamically partitionable server, the configuration of partition units that are assigned to a particular hardware partition can be changed while the system is running. Partition units can be added or replaced without restarting the instance of the operating system that is running on the hardware partition. Common operations include:

Hot Add

Adding a partition unit to a running hardware partition.

Hot Replace

Replacing a partition unit with an identical replacement partition unit that is already present in the system. Note that this is a single atomic operation that is not the same as a hot remove operation followed by a hot add operation.

Hot Remove

Removing a partition unit from a running hardware partition. Windows Server 2003 SP1 Enterprise Edition and Datacenter Edition support hot add of memory on x86-based, x64-based, and Itanium-based systems. Windows Server 2008 supports hot add of processors, memory, and I/O host bridges plus hot replace of processors and memory on x64-based and Itanium-based systems.

Features :

(a) Processors, memory, and I/O host Bridges can be hot added to a hardware partition while the system is running.

(b) Processors and memory can be hot replaced in hardware partition while the system is running.

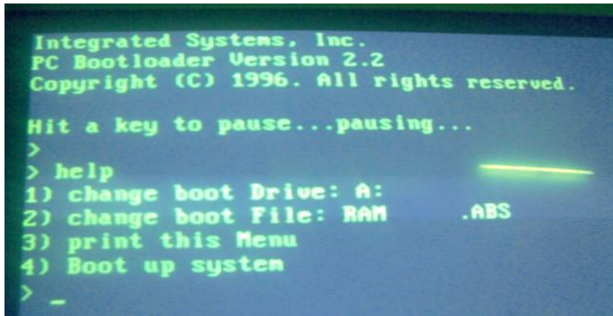
(c) Device drivers and applications can register to be notified of changes to the hardware configuration so that they can adjust their resource allocations accordingly.

(d) Existing applications should continue to run without modification. However, in order for an application to take advantage of any new hardware resources that are added to the hardware partition after the application has been started, the application must register for notification of changes to the hardware configuration.

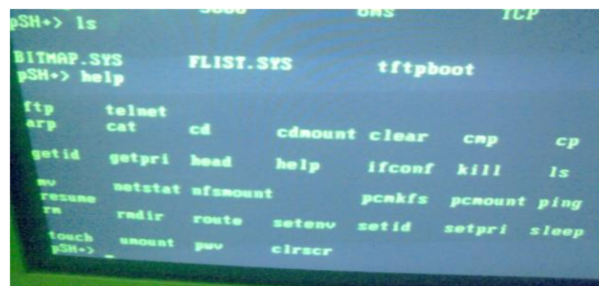
Circuit diagram

VI. OUT PUT

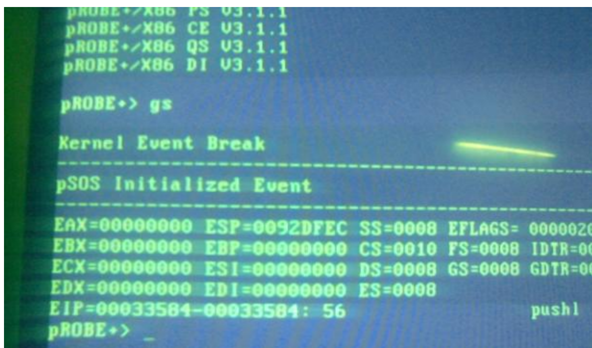
PSOS BOOTLOADER



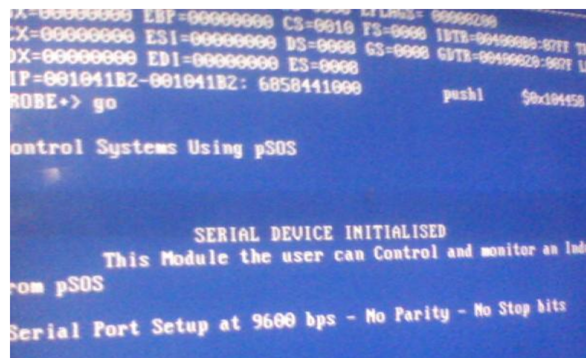
PSOS FILESYSTEM



PSOS INITIALISE

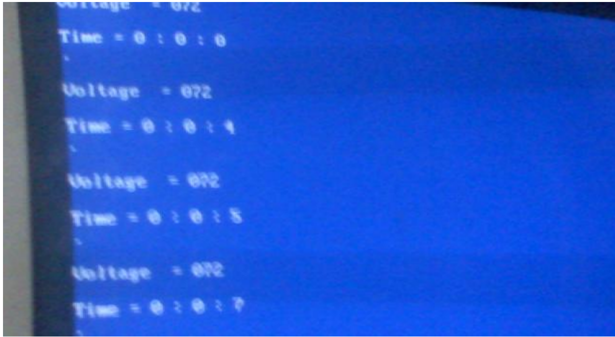


INITLIAISE TASK



Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

INPUT TASK



OUTPUT TASK

VII. CONCLUSION

We presented the design of RTFS, a real-time file system that complies fully with the file system standard. In addition, the RTFS also provides a POSIX compliant interface to support legacy code. RTFS provides a predictable temporal behavior, bounding the file operation times. RTFS implements the space and time partitioning in a rigorous manner and includes support for network centric operations, and a variety of mass storage devices. RTFS also supports metadata journaling on non-volatile memory devices to implement fast file updates and fast file system recovery on faults.Our effort at designing RTFS clearly demonstrates that it is possible to implement a file system that provides real-time performance as well as the space and time partitioning .We address the variable access latencies inherent to mass storage devices like hard disks and DVD/CD drives and make use of a two- level scheduling mechanism to guarantee and bound the access times to such devices. Our low-level scheduling technique for these storage devices additionally ensures that transfer rates to/from these devices are maximized. Although not discussed here, the basic design approach for RTFS makes it possible for us to handle mass storage implemented on RAM as well as non-volatile storage devices such as NVRAM or Flash memory.

REFERENCES

1. Avionic Application Software Standard Interface – ARINC Specification 653, Aeronautical Radio Inc., 1997.
2. ARINC 653 File System Standards Draft –Revision 5, version of Feb. 2005.
3. ARINC 653 File System Standards Draft – Revision 5, version dated June 8, 2005.
4. ARINC 653 File System Standards, discussions and comments from the meeting of March 1 to 3, 2005.
5. Ghose, K., Aggarwal, S., Vasek, "ASSERTS: A Toolkit for Real-Time Software Design, Development and Evaluation", in the Proc. of the 9-th Euromicro Real-Time Systems Workshop (available from the IEEE CS Press), 1997.
6. Bosch, P. and Mullender, S. J., "Real- time Disk Scheduling in a Mixed-Media File System". In Proc. RTSS-2000.
7. Shenoy, P. J., and Vin, H. M., "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", Master's Thesis, Univ. of Texas.
8. Gopalan, K., "Real-time disk scheduling using deadline sensitive scan", Technical Report TR-92, Dept. of Computer Science, State University of New York, Stony Brook, 2001.
9. Reuther, L. and Pohlack, M., "Rotational- Position- Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS)", in Proc. Real- Time System Symposium (RTSS), 2003.
10. Zhang, Z., and Ghose, K., "yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking", in Proc. of the USENIX Symposium on File Systems and Storage Technologies (FAST '03), 2003.