A Minimal API For Support of Real-Time Operations.

University of Columbia
Computer Science Department
Dirk Bridwell dnbridwell@earthlink.net
Jan Miller janalanmiller@uswest.net

Abstract

To address the growing need to develop applications that have real-time requirement, this paper suggests an operating system level API. The requirements for this API are derived from a survey of existing operating systems. The survey compares the popular operating system Windows NT/2000, two flavors of Linux (plain Linux and RTLinux), and one traditional Real-Time Operating System, pSOS+. The survey discusses real-time aspects of the operating systems' application programmers' interfaces (APIs) and underlying implementations. An API is proposed that meet the minimum required functionality necessary to support a real-time application.

Introduction

In his survey of real-time operating systems, Gopalan [2] states that the need for Operating Systems to support applications, which require real-time behavior, is growing. From the desktop to very specialized software/hardware combinations, developers are required to create systems that can satisfy both hard real-time and soft real-time requirements. As Yodaiken and Barabanov [5] indicate, design cycles can be so short, and projects are becoming more diverse in their demands. Developers want familiar and robust (API), tools, and programming environments to create their real-time applications. The developer's ability to integrate with their familiar tools is important, but popular operating systems do not have equal levels of support for real-time applications. Obenland [1] points out that since the design of an OS can have a significant impact on its ability to be used in a real-time system, implementation has to be considered along with the APIs provided.

A survey of the literature provides the necessary qualities of a real-time operating system and the requirements for a real-time API. The survey concentrated on four operating systems: Microsoft Windows NT, Linux, pSOS+, and RT-Linux. The operating systems in the survey were chosen for both popularity and varying approaches to the problem of real-time application support. Windows NT and Linux are both General Purpose Operating System. RT-Linux is a hybrid that adapts the standard Linux kernel to support real-time. pSOS+ is a commercial real-time Operating System. The operating systems were evaluated using the necessary qualities taken from the literature. The results are not unexpected. Windows NT and Linux might be desirable for soft real-time applications, but they do not provide adequate support for hard real-time applications. RT-Linux can support hard real-time with some caveats. Only pSOS+ has the necessary support for Real-Time operation.

The suggested real-time API is broken into three parts: Process Management, Interprocess Communication, and Memory Management. Process Management encompasses the creation and scheduling of multiple concurrent threads of control. Interprocess Communication provides synchronization and information sharing between processes. Memory Management is the allocation and organization of memory available to an application.

Necessary Qualities of a Real-Time Operating System

There are several qualities, which an operating system must have to support real-time applications.

Process Management
According to Obenland [1] an operating system must support multiple threads. The threads must be preemptible by the operating system. There must be a well-defined way to assign priorities to threads. The number of thread priorities should be sufficient to support many threads, each assigned a different priority. Support for 256 priority levels seems to be the consensus. The scheduler must ensure that threads that need to run are able to do so. A class of threads, Interrupt Handlers, must receive special consideration in scheduling. Yodaiken and Barabanov [5] state that an operating system must be able to quickly deal with interrupts.

Interprocess Communication
Any nontrivial system threads must share information. This can be accomplished through a variety of mechanisms. The simplest is shared memory. Because of the need to allow concurrent access to shared resources, an operating system must provide predictable synchronization mechanisms. The most basic is the standard mutex (or lock). A thread must be able to bound the time spent waiting for a lock. To support this, an operating system must have support for high-resolution clocks and timers. The accuracy of the timers will ultimately be dependant on the underlying hardware. An operating system must support priority inheritance to prevent priority inversion when a thread is waiting for a lock.

Memory Management
An operating system needs to supply predictable memory management. A large source of unpredictability is virtual memory. If a page fault is generated when accessing memory, a process is blocked for an unbounded period of time. In order to avoid this, virtual memory must either not be allowed or an application must be able to lock its allocated memory into RAM.

Operation System Survey

The operating systems considered in this survey have varying degrees of support for the necessary qualities of real-time systems.

Windows NT

Windows NT is a general purpose operating system. Due to its widespread acceptance, its use as a platform for Real-Time applications is unavoidable. Windows NT implements the Win32 API. Win32 is a diverse API with support for everything from low-level operating system services to graphical interfaces.

Process Management
Windows NT supports priority based preemptive scheduling. It supports 32 levels of priority with 32 being the highest. The priority specified when a thread is created is known as the base priority. The actual priority of a thread is not fixed. Windows NT will boost the priority of a thread if it has not had enough CPU time. Windows NT uses Deferred Procedure Calls (DPC's) to processes interrupts. DPC's are placed in a FIFO queue. Ramamritham, et al [7] claims that this disregards any associated priority. DPC's are preemptible by interrupts even if the interrupt is lower priority. Thus the time it takes to handle an interrupt is unpredictable.

Interprocess Communication
Win32 supplies mutexes, semaphores, queues and other synchronization and communication mechanisms. The mechanisms can have a timeout associated with them to make sure that they have bounded waiting times. According to the Win32 SDK documentation [9], the order in which threads will acquire a synchronization mechanism is not guaranteed. Thus, Ramaritham, et al [7] concludes Windows NT does not support priority inheritance. Adding to the unpredictability of waiting in Windows NT is the lack of high-resolution timers. Gopalan's [2] research indicates Windows NT can only support delays of 10's to 100's of milliseconds. Windows NT does provide a high-resolution clock using the windows multimedia timer [9], but the resolution is totally dependant on the hardware.

Memory Management
Processes in Windows NT operate in their own memory space. To accomplish this Windows uses a paged virtual memory system. While this is undesirable for Real-Time support, Pages can be locked into memory. They may still be swapped out if the process is inactive or if the window running the process is iconized. Timmerman and Monfret [8] indicate that the former is difficult to produce and the latter is unlikely to be an issue for Real-Time applications.

Linux

Linux is an open source Unix clone. As such, it is designed to be a general-purpose operating system. The bulk of Linux's real-time functionality is represented by the standards POSIX.1b, POSIX.1c, and SystemV APIs. Most of the discussion about Linux addresses the POSIX Real-Time standard and Linux's support for it.

It is important to remember that there is no bottom line with Linux or its APIs because it is constantly being developed. New API's are constantly being grafted on through patches and projects. All information supplied here is at best a snapshot of a version of the kernel.

Process Management
Linux supports preemptive prioritized scheduling. In order to comply with the POSIX, Linux defines thirty-two levels of priority. Threads may choose to be scheduled in a FIFO order and run to completion, or they may be scheduled in a round-robin fashion. According to Barabanov and Yodaiken [4], assigning the highest priorities to critical tasks does not help, This is partly because of the Linux "fair" time-sharing scheduling algorithm.

Epplin [3] states that the fundamental problem faced when attempting to graft POSIX.1b functionality onto Linux is the fact that Linux has a non-preemptible kernel. Since the kernel is non-preemptible, interrupts can be delayed.

Interprocess Communication
Linux does provide standard synchronization mechanisms. However, Linux fails to comply with the POSIX.1b spec. According to Epplin [3], the timer functions and POSIX.1b signals are not yet complete, and Linux does not implement the real-time semaphores or message queues. Linux can only supply precision of about 10 milliseconds using POSIX real-time functions. POSIX timers are only supported through patches to the kernel.

Memory Management
Linux like other general-purpose operating systems provides virtual memory. Barabanov and Yodaiken [4] point out that bringing requested pages back to RAM takes an unpredictable amount of time. This can be overcome since the POSIX memory locking facilities have been implemented. Garnett [6] states that by locking pages into memory and using the round-robin scheduler a certain degree of predictability is achievable. Unfortunately Linux is still not able to meet even moderately demanding real-time requirements.

RTLinux
RTLinux belongs to the class of operating systems that attempt to adapt a general-purpose operating system to handle real-time requirements. The RTLinux operating system works by emulating interrupt control for the Linux kernel. The Linux kernel simply runs as the lowest priority RTLinux process. Most services are still provided by the Linux kernel. The intention is to have the RTLinux kernel provide only the services that Linux cannot provide.

Process Management
The RTLinux scheduler is purely priority driven. The is simply ensures that the highest priority thread is scheduled to run. The run order of two standard threads at the same priority is undefined. RTLinux supports POSIX Pthreads API with an extension for threads to be scheduled based on a required period. The RTLinux documentation does not specify how many levels of priority it supports. Since it claims to support POSIX.1b threads it must support at least 32 priority levels.
RTLinux defines two types of interrupts: hard and soft. Only hard interrupts are appropriate for real-time applications since soft interrupts are handled just like Linux interrupts. This ensures that the only resource that is reliably shared is the CPU.

Interprocess Communication
RTLinux supports its own queue mechanism called RT_FIFO's and its own shared memory routines. It also supports POSIX mutexs, and semaphores. RT_FIFO's are part of the Linux kernel's memory and are never paged to disk. While communication with Linux threads is possible, it is generally not safe because the Linux kernel disables interrupts to provide synchronization.

Memory Management
RTLinux does not provide any dynamic memory for its threads. Each thread is loaded into its own address space. Yodaiken and Barabanov [5] say that this enforces the basic approach that more sophisticated tasks should be left to Linux processes. The lack of virtual memory ensures that page faults never occur.

pSOS+
The pSOS+ operating system is considered a traditional Real-Time Operating System. The major difference between pSOS+ and the general purpose operating systems is that pSOS+ will not attempt to provide unlimited resources. The number of operating systems resources is fixed at compile time. Any attempt to exceed the finite resources will generate errors. While pSOS+ does provide support for more common API's like POSIX and the standard C library, these are largely present for the sake of portability (information in this section comes from the pSOS+ manuals [10]).

Process Management
pSOS+ employs a priority-based, preemptive scheduling algorithm. Unlike Windows NT or Linux, it does not attempt to be fair, and will ensure that the task with the highest priority is running. The scheduler defines 256 priority levels. Level 256 is the highest. Since the pSOS+ kernel has no threads of its own, preemption only occurs when a thread makes a system call. Little is documented about how pSOS+ handles interrupts. Interrupt handlers are not allowed to use unbounded blocking operations, such as an indefinite wait on a mutex.

Interprocess Communication
As of version 2.5 pSOS+, supports many IPC and synchronization mechanisms including mutexes, semaphores, condition variables, message queue, etc. These are recent additions that are a vast improvement over previous releases. In order to perform synchronization, threads used to have to disable interrupts. With the new additions, pSOS+ added support for priority inheritance.
The method for waiting on queues or mutexes is either priority based or FIFO. The method is decided by the application. pSOS+ supports high resolution timers and clocks. The documentation claims its proprietary timing and scheduling algorithm guarantees constant time operations.

Memory Management
pSOS+ organizes memory into multiple regions. Regions are further broken into segments. The exact organization is left to the application. Only one region must be created and it is reserved for the operating system. This region is called special region 0. Memory may be allocated from this region through the use of the standard C/C++ routines. pSOS+ allows an application to manage its own memory. It provides no virtual memory management.

Suggested API

The suggested API comes from a synthesis of the surveyed operating systems. The essential services that are available in most of the operating systems are part of this API. Services that were present in one or two operating systems, but offered enhanced functionality are also present. Services that did not meet the reviewed guidelines have been left out. The API is minimalist. It attempts only to address needs of real-time applications. Other services would need to be included to make this a fully functional API. Higher-level functionality, such as C libraries and networking, have been ignored.

Process Management
The process management section allows for the management of concurrency. Here, applications can define and start different threads of control. It also allows applications to install interrupt services.

Suggested functions
ThreadCreate: Declares and starts a thread in the system. The application would need to specify priority, period, scheduling algorithm, and starting address. Return thread id.
ThreadWait: Blocks a thread until its next period of execution. Only used internally to the thread. Works from within current thread. Timing is set up when thread is created.
ThreadDestroy: Forcefully ends the thread. The scheduler will not consider the thread again.
ThreadJoin: Allows a thread to block until another thread has exited.
ISRCreate: Installs a handler for a specified interrupt. Application must specify start address, and priority.
ISRDestroy: Removes handler for specified interrupt. Must specify ISR id.

Suggested Implementation
The scheduler need not be complex. All that is needed is a priority based preemptive scheduler. The scheduler should not disable interrupts. For flexibility the scheduler should support at least 256 levels of priority. Both application installed and system level interrupt handlers should be as short as possible.

Interprocess Communication
The interprocess communication section defines mechanisms for synchronization and communication between threads.

Suggested functions
MutexCreate: Creates a new mutex. Options include using Priority Inheritance and type of waiting queue. The waiting queue may either be priority or FIFO.
MutexAcquire: Acquires the lock on this mutex. Must specify mutex id. May specify timeout.
MutexRelease: Releases the lock on this mutex. Must specify mutex id.

MutexDestroy: Removes a mutex from the system. Must specify mutex id.
CVCreate: Creates a new condition variable. Specify type of waiting queue, either priority or FIFO.
CVWait: Blocks a thread until the condition variable is signaled. May specify timeout. Must specify cv id.
CVSignal: Signals threads waiting on this condition variable. May specify to signal one thread or all threads. Must specify cv id.
CVDestroy: Removes a condition variable from the system. Must specify cv id.
QueueCreate: Creates a new FIFO queue. Queue depth may be fixed or variable. Options include priority inheritance, and type of waiting queue.
QueueEnqueue: Places a new item at the end of the queue. Must specify item and queue id. May specify timeout.
QeuueDequeue: Removes the item at the head of the queue. Blocks until an item is available. Must specify queue id. May specify timeout.
QeuueDestroy: Removes a queue from the system. Must specify queue id.
Sleep: Blocks the execution of a thread for a period of time. The time must be specified. Affects the current thread.

Suggested Implementation
It is important that the operation of the synchronization and communication mechanisms is predictable. The timers used must be accurate with very small amounts of jitter. The order in which locks are acquired must be deterministic.

Memory Management
The Memory Management section gives control to the application to manage its own memory with minimal involvement from the operating system.

Suggested functions
SegmentCreate: Reserve a new memory segment for use. Must specify range.
SegmentDestroy: Gives control of a memory segment back to the operating system. Must specify the segment id.
MemoryAllocate: Bind a block of memory to a variable. Must specify the size of the block and the segment id the block belongs to.
MemoryDeallocate: Unbind a block of memory. Must specify the block's starting address.

Suggested Implementation
The operating system should allow applications to control their own memory. This means that virtual memory should either not be implemented or the application must be able to lock pages into memory. Using the specified functions, the application should be able to construct its own memory management scheme.

Conclusions

The requirements for an operating system to support real-time operation are well understood. The most important issues address how deterministic the operating system is. Windows and Linux support only soft real-time because they are not deterministic enough. RTLinux support for hard real-time is limited if the non-deterministic Linux services are used. The most deterministic operating system, pSOS+ supports hard real-time.

A real-time API can provide most services that are present in a general purpose operating system. The most important factor in designing such an API is to give the application as much control as possible over how it is scheduled. Increasing control is accomplished by allowing the application to determine its priority and to choose how long it waits for resources. This lets the application bound the amount of time that it is blocked.

While the APIs of an OS are very important to a developer's decision to use a particular OS for his/her application, there are many factors, which should be considered before implementation begins. Each operating system compared in this survey fits a particular set of problems. Window's huge user install base makes it an attractive alternative. Linux and RTLinux share the advantages of open source, easy modification, and memory footprint scaling. Unpredictable Device drivers reduce the determinism of Windows and Linux. The limitations of RTLinux make it ideal only for small and simple hard real-time tasks in a Linux environment. Developers who want only hard real-time capabilities with predictable device drivers and more application control should consider pSOS+. When a small memory footprint is necessary, pSOS+ is also a good candidate.

Future work would include implementing an operating system with the suggested API. This prototype OS could be used for experiments in improving algorithms and fine-tuning the suggested API. A follow up to this paper would address admission control and negotiations with the operating system a desired level of service.

References:

[1] Obenland, Kevin M. POSIX in Real-Time. Embedded Systems Programming, Vol. No. 4, April 2001.

[2] Kartik Gopalan, Real-Time Support in General Purpose Operating Systems, Research Proficiency Exam Report, Dept. of Computer Science, State University of New York, Stony Brook, NY, January 2001.

[3] Epplin, Jerry. Linux as an Embedded Operating System . EmbeddedSystems Programming 10(10), October 1997.

[4] Barabanov, Michael and Yodaiken, Victor. Linux Means Business: Introducing Real-Time Linux. Linux Journal. February 01, 1997. http://www.linuxjournal.com/article.php?sid=0232.

[5] Victor Yodaiken and Michael Barabanov. A Real- Time Linux. In Proceedings of the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA, January 1997. The USENIX Association.

[6] Garnett, Nick. EL/IX Base API Specification DRAFT - V1.2. Red Hat Inc. September 18, 2000.

[7] Ramamritham, Krithi, Shen, Chia, González, Oscar, Sen, Subharata, and Shirgurkar, Shreedhar. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendation. In IEEE Real-Time Technology and Applications Symposium. 1998.

[8] Timmerman, Martin, and Monfret, Jean-Christophe. Windows as a Real-Time OS. Real-Time Magazine p6-13, 2Q97 .

[9] Microsoft Platform Software Development Kit. January 2001.

[10] pSOSystem System Concepts. Integrated Systems Inc. 1999.