Analysis real-time scheduling on the Linux kernel and RTAI

Rajesh J. Nagar*[1], Ramesh Joshi**[2]

*Department of Information Technology, Hasmukh Goswami College of Engineering, Ahmedabad
Gujarat Technological University, India
[1]rajesh.nagar.it@hgce.org
**Department of Information Technology, Shantilal Shah College of Engineering, Bhavnagar
Gujarat Technological University, India
[2]arjoshi07@gmail.com

Abstract

Linux kernel used in the many open source distributions is the normal/default kernel which doesn't support real time scheduling. If an embedded developer wants to compare the scheduling policies of Linux to a real time operating system it is more useful to compare RTOS performance to a version of Linux that does have real-time features. Fortunately, in addition to this default kernel, there is also available a Real-time kernel version that supports a real-time scheduling policy. In this article and in the code examples that are included, an effort is made to compare the real time operations of standard and real-time Linux with normal RTOS operation and evaluate the differences and similarities. We have used RTAI which is real-time extension of Linux kernel for real-time performance.

Keyword: RTOS, RTAI, Scheduling in Linux, FIFO Policy in RTOS

1. Introduction

Normal Linux Kernel is a pre emptive kernel but not real time, of course. In most multithreading environments (also called multitasking), a pre emptive kernel allows the thread that has higher priority to receive longer time on the processor. And, conversely a lower priority thread will have less time with the processor.

However, in the normal kernel, no particular thread can monopolize the services of the resident processor all the time, no matter what its priority. So, programs will never hang up even if an arbitrary thread of the program goes into a "forever loop."

Conversely, with almost any real time OS (such as Free RTOS, Micrium uC/OS, and ThreadX), the kernel supports both pre emption and real-time features. This means that a thread (also called a Task) can run forever when the following conditions are met: (1) it is not blocked by synchronized resource (I/O block, Mutex, Semaphore…) or (2) it isn't pre empted by threads which may have equal or higher priority.

2. Scheduling Policy

In this section we show different scheduling policy. Figure 1 below shows the differences between the thread scheduling policy of normal Linux and the thread scheduling policy of a RTOS.

In Figure 1, we assume that Thread1 has priority 2 and is higher than Thread2 which has priority 1. In the RTOS scheduler, Thread1, with higher priority, always runs. Thread2 with lower priority never has a chance to run. By comparison the Normal Linux Scheduler does exactly the opposite, bringing up both Thread1 and Thread2 to run. Thread1 with higher priority will have longer time to run as compared with Thread2 which has lower priority.

2.1. Schematic Diagram

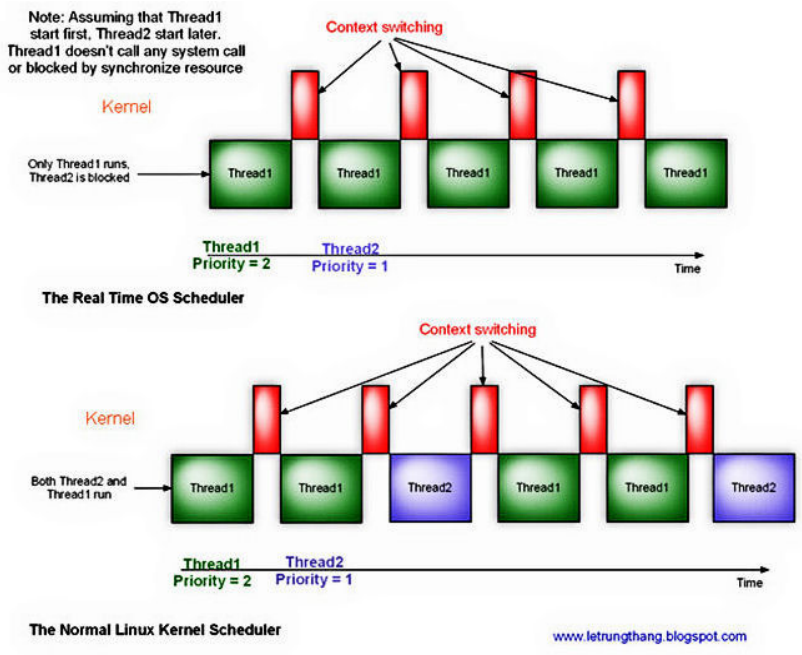


Figure 1: Figure 1. Real Time OS and normal Linux kernel scheduler

2.2. Real Time Linux kernel versus RTOS

In a real time kernel version of Linux, the scheduler has three (3) scheduling policies: Normal, FIFO and Round Robin. In the Normal scheduling policy, a thread will be stopped (suspended) when one of three conditions occurs:

1. It is blocked by an accessing synchronize resource (I/O block, mutex, semaphone...)
2. It volunteers to give up control of processor (call sleep () or pthread_yield()).
3. The Scheduler suspends the thread when its running time exhausted. The running time depends on each thread's priority, as noted in Figure 1. Normal real time scheduling policy is same as the default scheduling policy of normal kernel as described earlier.

With the FIFO scheduling policy, a thread will be stopped (suspended) when one of three conditions occurs:

1. It is blocked by accessing synchronize resource (I/O block, mutex, semaphore...)
2. It is pre empted by a higher priority thread.
3. It volunteers to give up control of processor (call sleep() or pthread_yield()).

In the Round Robin scheduling policy, a thread will be stopped (suspended) when one in four following conditions occurs:

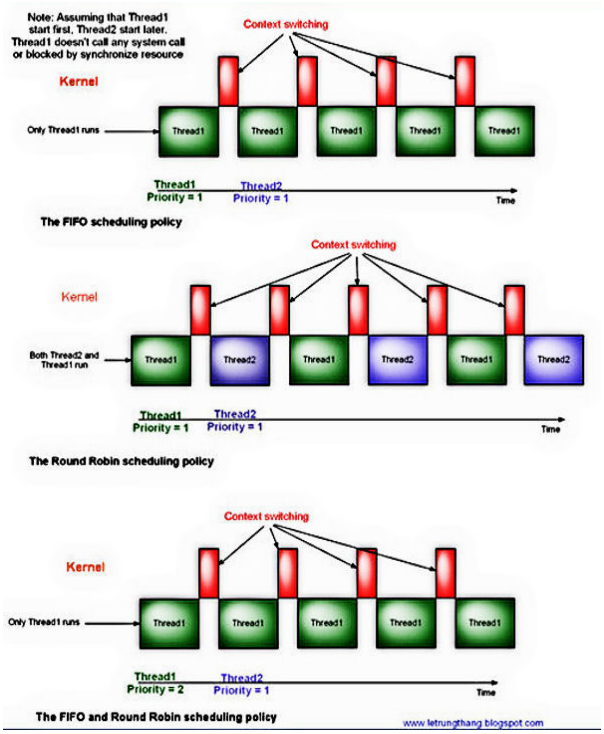
1. It is blocked by accessing synchronize resource (I/O block, mutex, semaphore...)
2. Or it is pre empted by a higher priorit thread.
3. Or it volunteers to give up control of processor (call sleep() or pthread_yield())
4. Or its Time slice expired.

Most RTOSes embed Round Robin scheduling policy in their scheduler.

In Figure 2 below, we have four situations: First, Thread1 and Thread2 are of equal priority and run using the FIFO policy. Second, Thread1 and Thread2 are of equal priority and run on the Round Robin policy. Third, priority of Thread1 is higher priority of Thread2 and they run using the FIFO

policy. Fourth, the priority of Thread1 is higher priority of Thread2 and they run using the Round Robin policy.

We also assume that Thread1 starts first, then Thread2 starts later and during running time, Thread1and Thread2 don't make any system calls or are blocked by any synchronized resource (I/O, Mutex, Semaphore…). The result: the only differences that affect real time performance between the FIFO and Round Robin policies is when Thread1 and Thread2 have same priority. If the low priority tasks exhaust the low priority memory pool, they must wait for memory to be returned to the pool before further execution [1].



3. Conclusion

Viewed in this context, the typical RTOS scheduler is just a special case of real time Linux scheduler, or in other words, the RTOS scheduler is the real time Linux scheduler running with the Round Robin. To achieve deterministic output with low latency one should must use RTAI kind of RTOS extension or proprietary RTOS.

5. References

[1] S.R.Ball, Embedded Microprocessor Systems, Second edition, Butterworth-Heinemann, 2000.
[2] M.Bunnell, "Galaxy White Paper," http://www.lynx.com/lynx_directory/galaxy/galwhite.html
[3] G.D.Carlow, "Architecture of the Space Shuttle Primary Avionics Software System,"CACM, v 27, n 9, 1984.
[4] CompactNET, http://www.compactnet.com.
[5] H.Gomaa, Software Design Methods for Concurrent and Real-time Systems, First edition, Addison-Wesley, 1993.
[6] S.Heath, Embedded Systems Design, First edition, Butterworth-Heinemann, 1997.
[7] http://www.rtos4voip.com/index.html
[8] IEEE. Information technology--Portable Operating System Interface (POSIX)-Part1:System Application: Program Interface (API) [C Language], ANSI/IEEE Std 1003.1,1996 Edition.
[9] Jbed RTOS, http://www.esmertec.com
[10] E.Klein, "RTOS Design: How is Your Application Affected?," Embedded Systems Conference, 2001.

