Optimizing Linux Kernel for Real-time Performance On Multi-Core Architecture

P. Bala Subramanyam Raju *bsr3011@gmail.com*
Research Scholar,
S.V University,
Tirupathi Chittoor (Dt) AP, India

P. Govindarajulu *PGovindarajulu@yahoo.com*
Professor, Dept. of Computer Science,
S.V University,
Tirupathi Chittoor (Dt) AP, India

Abstract

Linux kernel developed and distributed in open source doesn't support for Hard Real-time scheduling. The open source Linux kernels are designed in time sharing manner to obtain maximum throughput. With this, Linux Operating System is considered to be an OS, which is not supporting Real-Time Applications, natively it has some features, already included in the mainstream to provide real-time support. There are certain modified Linux kernels like RTLinux, Symbian OS, Nucleus OS, Lynx OS and Fusion RTOS [1] which are explicitly designed for hard Real-Time support [2]. These specially designed Real-Time Linux kernels is mostly targeted for special hardware's like embedded systems, robots, safety critical etc. ,very few kernels for general purpose. Most of these kernels are be available as proprietary or closed, excluding a very few and not suitable for all hardware architecture's.

Now a days Real-Time Performance has become universal requirement for computer games, multimedia systems, household monitoring and controlling appliances. So the general purpose Linux kernel needs to be optimized, to achieve Real-time performance to meet the user expectations. This paper tries to extract real-time performance from general kernel and suggest some techniques to optimize Linux kernel to meet real-time deadlines.

Keywords: Kernel, Embedded Systems, Deadline, Real-time, Scheduler, Hyper Threading.

1. INTRODUCTION

Multi-core processor, delivering high computing power with reduction in hardware cost. This reduction in the hardware cost helps large number of people to purchase high performance computers. A normal user can run special types of applications like robot controller, applications collecting data from physical sensors, which are real time in nature and are not supported by open source Linux. In order to achieve real-time response, the General Purpose Linux kernel has modified by adding two real-time scheduling policies as shown in figure 1.1.

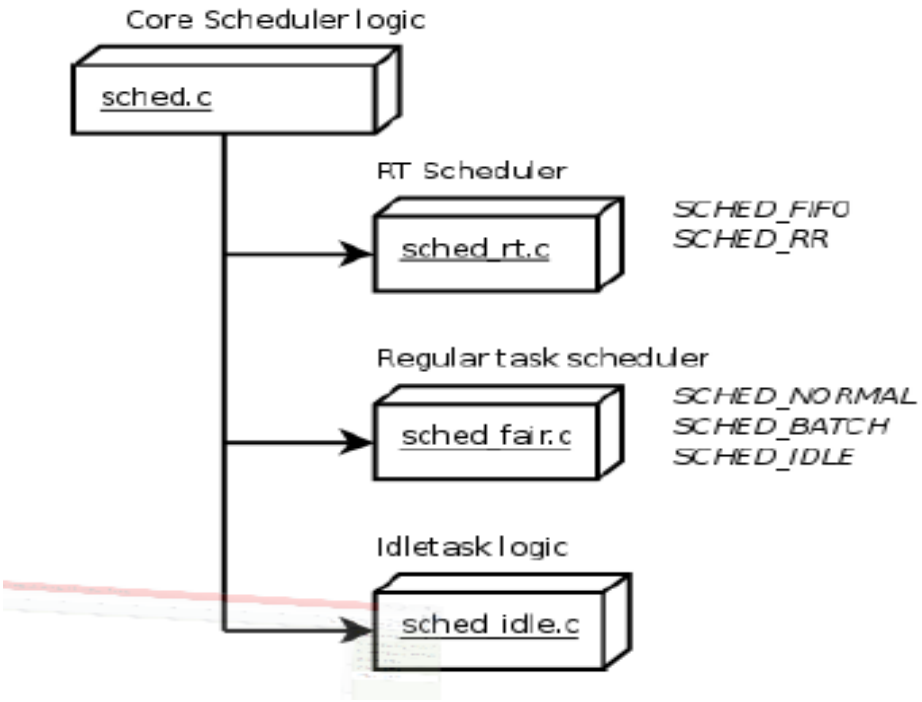

FIGURE1.1: Showing various Scheduling classes and policies in General Purpose Linux Kernel 4.0.

A real time system can be defined [3][4][5][6] as a *"system capable of guaranteeing timing requirements of the processes under its control".*

The following goals should be considered in scheduling a real-time system:

- Meeting the timing constraints of the system
- Obtaining a high degree of System utilization while satisfying the timing constraints
- Reducing the cost of context switches caused by preemption
- Reducing the communication cost in real-time distributed systems

Even after adding new scheduling classes and polices, the kernel needs to be fine-tuned to reach the goals of real-time system from multicore systems.

The remainder of the paper is organized as follows Section 2 discuss the previous work; section; Section 3 presents the hardware and software environment details, Section 4 describes an experimental performance evaluation; Section 5 proposes Optimization technique's. Section 6 gives implementation and experimental Results; Section 7 concludes the paper with future work.

2. PREVIOUS WORK

This section gives the overview of the research work carried out related to the performance improvement of Real-Time Systems.

Chenyang Lu, Xiaorui Wang and Xenofon Koutsoukos [7] proposed an approach to extend Quality of Service (QoS) from single processor to Distributed Real-Time Systems by a model predictive control approach. Utilization control is formulated as a multi-variable constrained optimization problem, second a dynamic model is established to formally characterize the coupling among multiple processor due to end to end tasks and practical constraints. MIMI model predictive controller is designed to control the utilization of multiple processors simultaneously. Finally stability analysis is performed to establish statistical guarantees on desired utilization

despite the uncertainty introduced by variation in task execution times. Simulation results demonstrate it can provide robust utilization guarantees when task execution times are significantly overestimated and change dynamically at run-time.

David Beal [8] an engineer of Freescale semiconductor, has specified that standard Linux kernel include enhanced schedulers, virtual memory, shared memory, POSIX Timers, Real-time Signals, POSIX IO,POSIX threads, Low Latency and many features that make Linux suitable for challenging real-time products and applications. Nat Hillary [9] of Freescale Semiconductor, presented methods for designing, measuring, improving the performance of real-time systems. Finally he concluded that real-time software is not something that can be done in a single step. Meeting required performance criteria can be obtained by careful consideration between the system and its environment needs. High fidelity software performance measurements may be achieved by using a combination of source code instrumentation and hardware.

Suresh Siddha,Venkatesh Pallipadi and Asit Mallick [10] proposed new scheduler optimization for Linux Kernel 2.6 for Chip Multi Processing (CMP). They discussed about generic OS Scheduler optimization opportunities that are appropriate in CMP environment. Henrik Austad, of Norwegian University of Science and Technology has introduced a new pfair algorithm for real-time tasks in the linux kernel on multi-core system. This scheduler will handle real-time task that cannon miss a deadline and is planned to be placed on top of the RT preemption patch. Due to problems faced during integration process, a fully functional scheduler has not been implemented.

Swati Pandit and Rajashree Shedge [11] has presented no of real-time scheduling algorithms that are suitable for simple uniprocessor and highly sophisticated multi-core processor. This paper also discusses the static, dynamic and hybrid priorities of a process. Finally conclusion shows that Instantaneous utilization factor scheduling algorithm gives better result in uniprocessor scheduling algorithms and Modified Instantaneous utilization factor scheduling algorithm gives better context switching, response time and CPU utilization as compared to previous scheduling algorithms.

Rohan R. Kabugade, S. S Dhotre, S H Patil [12] presents a modified algorithm named MOFRT (Modify O (1) For Real-Time) and Just-In-Time (JIT) based on the Linux kernel 3.2 to improve the Queue Management for Real time Tasks. Though, some of these algorithms have not been implemented since it is very hard to support new scheduling algorithms on nearly every operating system. However the previous works does not try to explore and fine tune the existing real-time supporting features included in the Linux Kernel Scheduler.

3. HARDWARE & SOFTWARE DETAILS

Processor	Intel® Core™ i7-4770k[5]
No of Cores	4
No of Threads	8
Base Frequency	3.5 GHz
Turbo Frequency	3.9 GHz
Intel® Smart Cache	8 MB
RAM	8 GB/1600 MHz

TABLE 1.1: Showing the hardware details.

- LINUX KERNEL 4.0-generic [14]
- LINUX MINT OS [15]
- TERMINATOR

- GCC COMPILER
- HTOP
- STRACE

4. EXPERIMENTAL PERFORMANCE EVALUATION

The goal of this experiment is to evaluate the performance of real-time scheduler included in the kernel 4.0. The program is designed to create a load to a multicore processor; and evaluate how far the existing scheduler supports for real-time programs in heavy load and normal situations, irrespective of other delays like data transfers, IO read & writes Network issues etc.

The algorithm of **"Load.C"** is as follows:

Step 1 : Start
Step 2 : repeat the following until true
step 2.1 : Stime=Read System time
step 2.2 : print 'Start time is:" stime
step 2.3 : j=0;
step 2.3 : Repeat the following steps until j<100
step 2.3.1 :i=0
step 2.3.2 :Repeat the following steps until i<1000000
step 2.3.2.1 : sum =sum +i;
step 2.3.2.2 : i=i+1
step 2.3.3 :i=0
step 2.3.4 :Repeat the following steps until i<1000000
step 2.3.4.1 : sum =sum +i;
step 2.3.4.2 : i=i+1
step 2.3.5 :i=0
step 2.3.6 :Repeat the following steps until i<1000000
step 2.3.6.1 : sum =sum +i;
step 2.3.6.2 : i=i+1
step 2.3.7 :i=0
step 2.3.8 :Repeat the following steps until i<1000000
step 2.3.8.1 : sum =sum +i;
step 2.3.8.2 : i=i+1
step 2.3.9 :i=0
step 2.3.10 :Repeat the following steps until i<1000000
step 2.3.10.1 : sum =sum +i;
step 2.3.10.2 : i=i+1
step 2.3.11 :i=0
step 2.3.12 :Repeat the following steps until i<1000000
step 2.3.12.1 : sum =sum +i;
step 2.3.12.2 : i=i+1
step 2.3.13 :i=0
step 2.3.14 :Repeat the following steps until i<1000000
step 2.3.14.1 : sum =sum +i;
step 2.3.14.2 : i=i+1
step 2.3.15 :i=0
step 2.3.16 :Repeat the following steps until i<1000000
step 2.3.16.1 : sum =sum +i;

step 2.3.16.2 : i=i+1
step 2.3.17 :i=0
step 2.3.18 :Repeat the following steps until i<1000000
step 2.3.18.1 : sum =sum +i;
step 2.3.18.2 : i=i+1
step 2.4 : etime=read system time
step 2.5 : print 'End time is:"etime
step 2.6 : 'The Loop used :' etime-stime 'seconds'
step 3 move to step 2
step 4 stop

The designed program has been executed with two different schedulers and priority. One with RT class, FIFO scheduler and highest priority of 99 using the command “chrt –f 99. /FIFO”. Other has executed with Fair Scheduler class and normal priority using the command “. /a.out”. The results are tabulated below:

Real –Time Threads	Normal Threads	Total No of Threads	Real-time Thread Failures/Sec
1	0	1	0
1	1	2	0
1	2	3	0
1	3	4	0
1	4	5	2
1	5	6	5
1	6	7	5
1	7	8	5

TABLE 1.2: showing execution results of Load.C.

The maximum time taken by the specified system to complete the loop execution is 2 seconds. The above results show that Linux Kernel supports real-time performance by default, until no of threads is equal to no of physical cores in the system without any deadline failure .After the no of threads increases than physical cores the system performance starts degrading.

The figure 1.2 shows the execution of “Load.c” program using terminator.

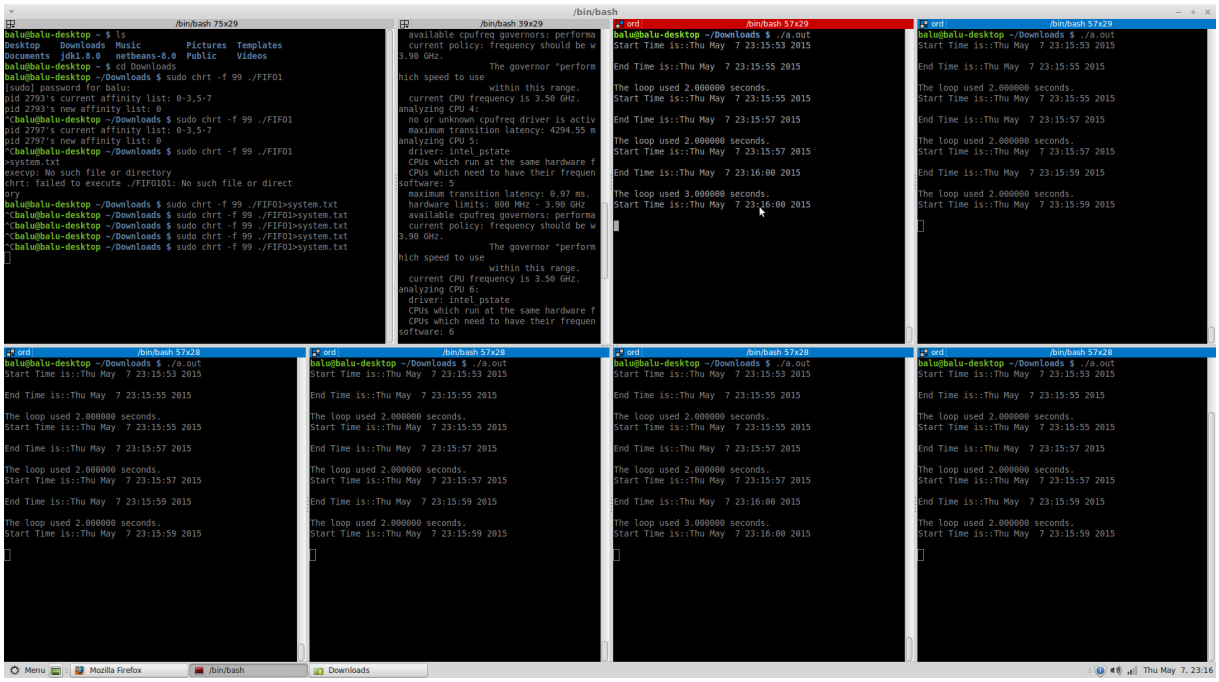

FIGURE 1.2: showing real-time thread and 7 normal thread execution.

5. PROPOSED OPTIMIZATION TECHNIQUE’S AND IMPLEMENTATION

There Linux kernel needs to be fine-tuned to support real-time environment in heavy load situations ,because the default kernel fully supports multitasking to increase the overall throughput buy using completely fair scheduling class the optimization techniques are specified below.

5.1 No Force Preemption

Preemption is one of the bottlenecks to real-time performance, because the kernel will preempt the thread non-voluntarily irrespective of program requirements and priorities, in order to support completely fair scheduling class. The default Linux kernel available in General Public License is built in way that the kernel can preempt the thread non-voluntarily. To improve real-time performance the Linux kennel needs to rebuild and update boot loader in a way that the kernel should not force thread to preempt. The procedure is explained below.

1. Download latest kernel from www.kernel.org[16]
2. Install git-core,libncurses5-dev tools required to build Linux kernel
3. Configure the options required using the $ make menuconfig
4. In menu configuration options select processor type and features-> preemption model Select “**No forced preemption (server)** “model shown in fig 1.3
5. Then build using make command
6. Install modules using command **$sudo make modules-install**
7. Install kernel using **$sudo make**
8. Update system configuration using **$ sudo update-initramfs-c-k 4.0**
9. Update boot loader **$ sudo update-grub**

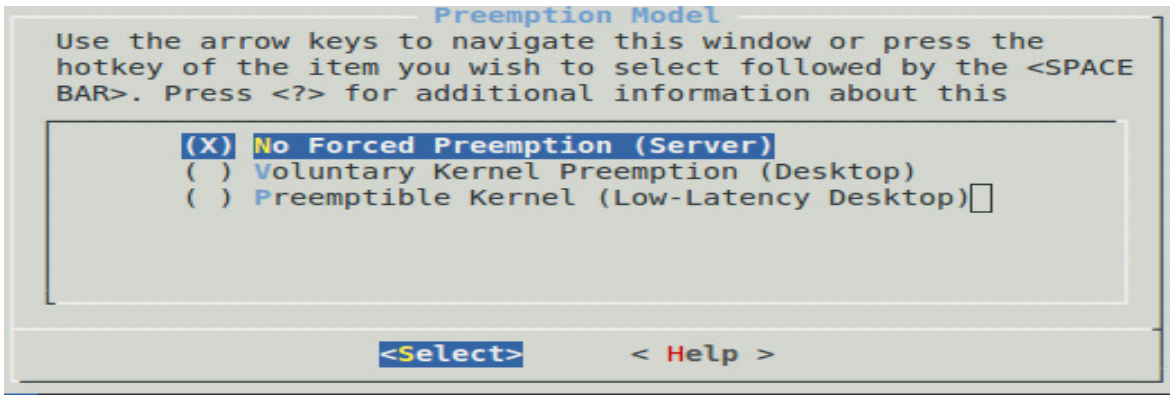

FIGURE 1.3: Showing the setting the preemption model during kernel building.

5.2 Stop Non-Voluntary Context Switching

The default Linux kernel will move thread from one core to other, in order to load balance the execution cores. This context switching takes CPU time to move executing thread from one core to other, by blocking the execution, which results in the execution delay leads to real-time failure. This can be avoided by setting the allowed CPU list to one specific core using the command **"taskset"**. The figures 1.4 and 1.5 shows the thread allowed CPU's list before and after setting CPU affinity. This can be viewed by using a command **"cat /proc/pid/Status"**,where pid is processID of a thread that needs to bind to a processor[17][18][19].

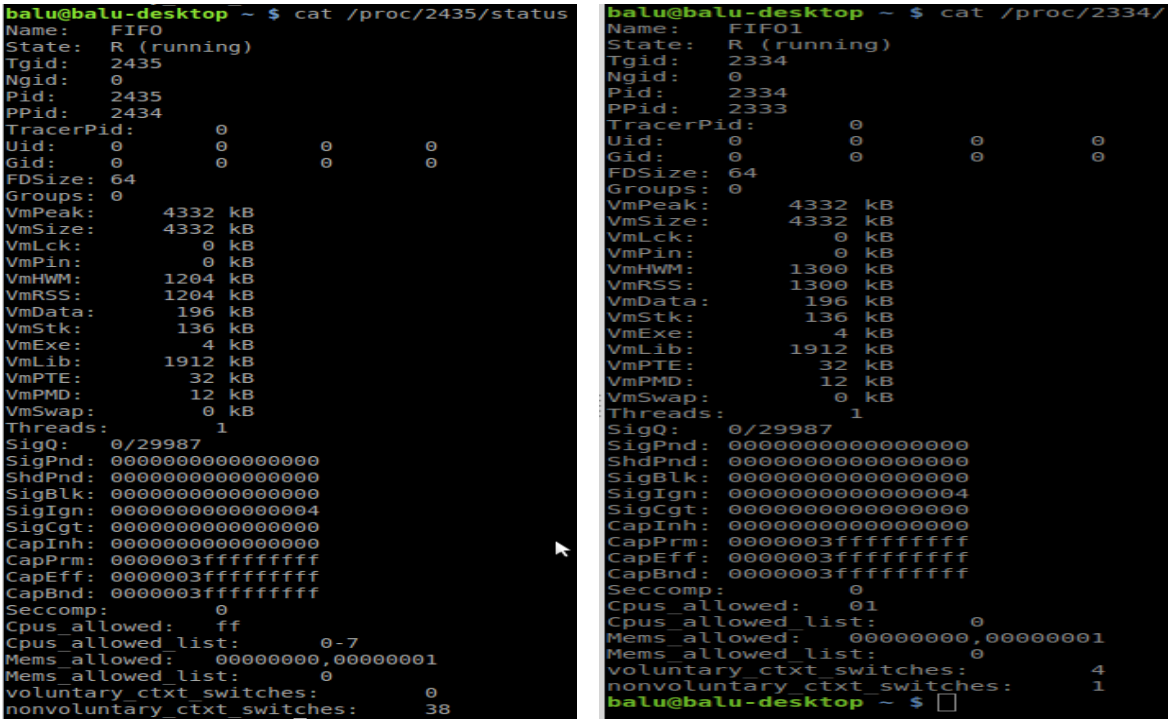

FIGURE 1.4 & 1.5: Showing process details before and after setting CPU affinities list and reduced context switches.

5.3 Hyper Threading (HT)

Intel Hyper-Threading Technology (Intel® HT Technology) is a technology used by some Intel microprocessors that allows a single microprocessor to act like two separate processors to the operating system and the application programs that use it. HT Technology utilizes resources

more efficiently .As a performance feature, it also increases processor throughput, improving overall performance on threaded software. Figure 1.6 showing the difference between Multiprocessor and Hyper threading processor [22][23][24].

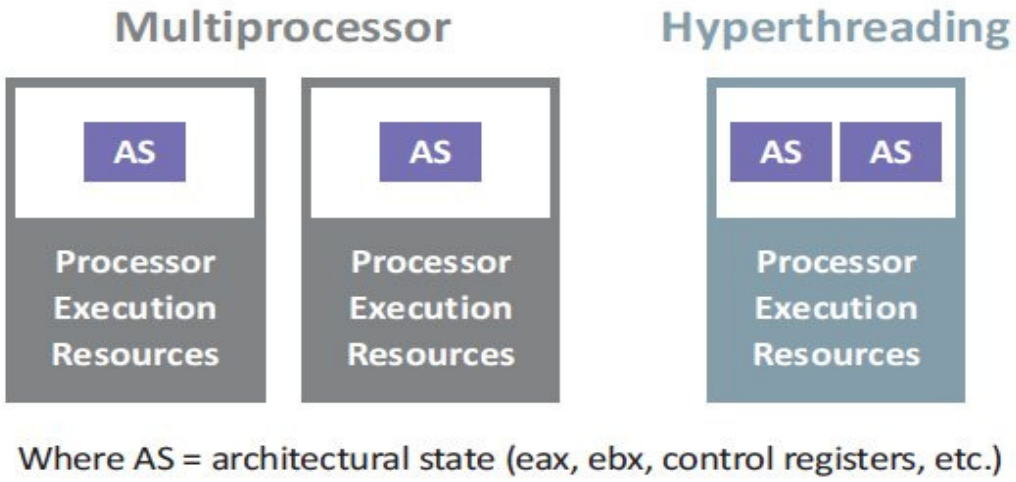

FIGURE 1.6: showing technological difference between multiprocessor and HT enabled processor.

This HT Technology becomes a major drawback for real-time performance because, the two threads compete for execution resource on single execution unit, and this results in sharing the CPU cycles between the two threads. Sharing real-time thread CPU cycles results in deadline failure, due to less execution time. To utilize 100 percent CPU cycles for real-time thread, the other thread on the core needs to be disabled. This can be achieved using the command **"echo 0 | sudo tee /sys/devices/system/cpu/cpu4/online",** here core no 4 has disabled. The figure1.7 showing the cpu4 disabled on quad core 8 threaded processor.

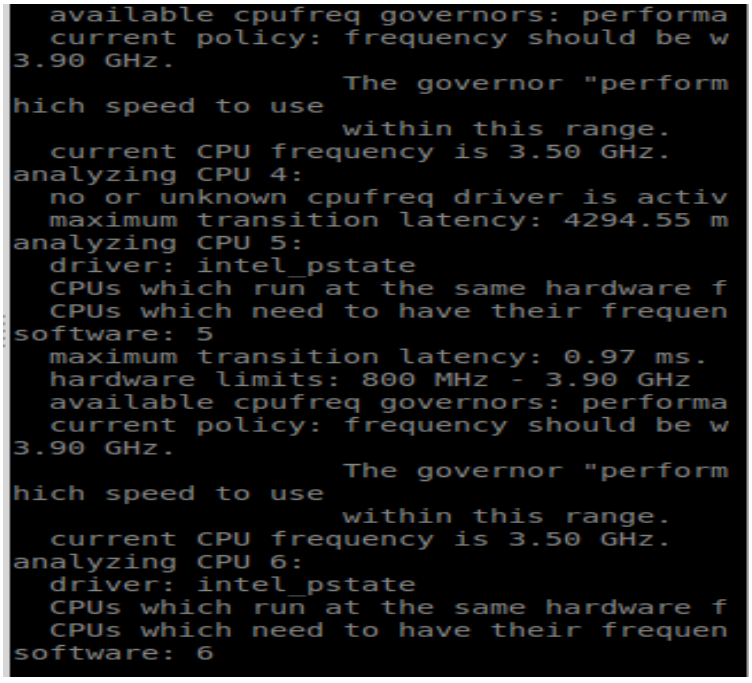

FIGURE 1.7: showing the cpu4 disabled on quad core 8 threaded processor.

6. EXPERIMENTAL RESULTS AFTER OPTIMIZATION

After fine tuning the operating system and hardware, the program also needs to change according to the requirements for real-time. The changed algorithm is shown below.

Step 1 : Start
Step 2 : Read current process ID from kernel
tid= getpid();
Step 3 : Set the current thread to execute on CPU0 using taskset –cp 0 tid
Step 4: Disable core 4 by executing the command as super user
"echo 0 |sudo tee /sys/devices/system/cpu/cpu4/online
Super user password
Step 5 : repeat the following until true
step 5.1 : Stime=Read System time
step 5.2 : print 'Start time is:" stime
step 5.3 : j=0;
step 5.3 : Repeat the following steps until j<100
step 5.3.1 :i=0
step 5.3.2 :Repeat the following steps until i<1000000
step 5.3.2.1 : sum =sum +i;
step 5.3.2.2 : i=i+1
step 5.3.3 :i=0
step 5.3.4 :Repeat the following steps until i<1000000
step 5.3.4.1 : sum =sum +i;
step 5.3.4.2 : i=i+1
step 5.3.5 :i=0
step 5.3.6 :Repeat the following steps until i<1000000
step 5.3.6.1 : sum =sum +i;
step 5.3.6.2 : i=i+1
step 5.3.7 :i=0
step 5.3.8 :Repeat the following steps until i<1000000
step 5.3.8.1 : sum =sum +i;
step 5.3.8.2 : i=i+1
step 5.3.9 :i=0
step 5.3.10 :Repeat the following steps until i<1000000
step 5.3.10.1 : sum =sum +i;
step 5.3.10.2 : i=i+1
step 5.3.11 :i=0

step 5.3.12 :Repeat the following steps until i<1000000
step 5.3.12.1 : sum =sum +i;
step 5.3.12.2 : i=i+1
step 5.3.13 :i=0
step 5.3.14 :Repeat the following steps until i<1000000
step 5.3.14.1 : sum =sum +i;
step 5.3.14.2 : i=i+1
step 5.3.15 :i=0
step 5.3.16 :Repeat the following steps until i<1000000
step 5.3.16.1 : sum =sum +i;
step 5.3.16.2 : i=i+1

step 5.3.17 :i=0
step 5.3.18 :Repeat the following steps until i<1000000
step 5.3.18.1 : sum =sum +i;
step 5.3.18.2 : i=i+1
step 5.4 : etime=read system time
step 5.5 : print 'End time is:"etime
step 5.6 : 'The Loop used :' etime-stime 'seconds'
step 6 move to step 2
step 7 stop

The table 1.3 shows the execution results after fine tuning the system for real-time performance.

Real –Time Threads	Normal Threads	Total No of Threads	Real-time Thread Failures/Sec
1	0	1	0
1	1	2	0
1	2	3	0
1	3	4	0
1	4	5	0
1	5	6	0
1	6	7	0
1	7	8	0

TABLE 1.3: showing execution results of real-time and normal threads.

7. CONCLUSION AND FUTURE WORK

The results show that after fine tuning the Linux kernel in consideration with hardware, it is possible to extract real-time performance from generally available open source Linux kernel.

This paper doesn't on concentrate on implementation of fully non-preemptible kernel. This experiment results in delay for normal priority process. If the number of real-time process increases and starts disabling the cores then only real-time process will execute until the hardware supports and normal process will block execution leads to imbalance execution.

8. REFERENCES

[1] Wikipedia Internet:www.en.wikipedia.org/wiki/List_of_real-time_operating_systems [May 10, 2015].

[2] Embedded Internet: www.embedded.com/design/operating-systems/4371651/9/Comparing-the-real-time-scheduling-policies-of-the-Linux-kernel-and-an- RTOS [May 10, 2015].

[3] Wikipedia Internet: www.en.wikipedia.org/wiki/Real-time_operating_system [May 10, 2015].

[4] Peter wurmsdobler "Real Time Linux Foundation, Inc.". Internet : www.realtimelinuxfoundation.org/ [May 10, 2015].

[5] Fernando S. Schlindwein “Real-time DSP” Internet:www.le.ac.uk/eg/fss1/real%20time.htm. [May 10, 2015].

[6] Kanaka Juvva “Real-Time Systems”Internet: www.users.ece.cmu.edu/~koopman /des_s99/real_time/. [May 10, 2015].

[7] Chenyang Lu, Xiaorui Wang, Xenofon Koutsoukos,” End-to-End Utilization Control in Distributed Real-Time Systems”, Distributed Computing Systems, 2004. Proceedings. 24th International Conference, 2004.

[8] David Beal,”Linux® As a Real-Time Operating System” , Freescale Semiconductor, Document Number: SWVERIFICATIONWP Rev. 0 11/2005.

[9] Nat Hillary,” Measuring Performance for Real-Time Systems” , Freescale Semiconductor, Document Number: GRNTEEPFRMNCWP Rev. 0 11/2005.

[10] Suresh Siddha, Venkatesh Pallipadi,” Chip Multi Processing aware Linux Kernel Scheduler”, Linux Symposium, Volume 2, 2006.

[11] Swati Pandit and Rajashree Shedge,” Survey of Real Time Scheduling Algorithms ” IOSR Journal of Computer Engineering e-ISSN: 2278-0661, p- ISSN: 2278-8727 Volume 13, Issue 2 (Jul. - Aug. 2013), pp 44-51,

[12] Rohan R. Kabugade, S. S Dhotre, S H Patil,” A Study of Modified O(1) Algorithm for Real Time Task in Operating System”, Sinhgad Institute of Management and Computer Application NCI2TM: 2014 ISBN: 978-81-927230-0-6.

[13] www.ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz/[May.10, 2015].

[14] Internet: www.kernel.org,[May.10, 2015].

[15] Linux Mint Internet:www.linuxmint.com//[May.10, 2015].

[16] Lakshmanan Ganapathy,” How to Compile Linux Kernel from Source to Build Custom Kernel” Internet: www.thegeekstuff.com/2013/06/compile-linux-kernel/ June 13, 2013 [May. 10, 2015].

[17] Robert Love,”CPU Affinity” Internet: www.linuxjournal.com/article/6799 [May.10, 2015].

[18] Internet: www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html[May.10, 2015].

[19] Internet: www.linux.die.net/man/1/taskset[May.10, 2015].

[20] Alexander Sandler, April 15, 2008 “SMP affinity and proper interrupt handling in Linux” Internet: www.alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux[May.10, 2015].

[21] Sandeep Krishnan on January 27, 2014,” Introduction to Linux Interrupts and CPU SMP Affinity”, Internet: www.thegeekstuff.com/2014/01/linux interrupts/ [May.10, 2015].

[22] www.intel.in/content/www/in/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html [May.10, 2015].

[23] http://whatis.techtarget.com/definition/Hyper-Threading[May.10, 2015]

[24] Internet : www.doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.taskscheduler.html[May.10, 2015].