CHAPTER-1

INTRODUCTION

The small high priority synchronized kernel operation between the hardware and standard Linux makes standard Linux instantaneous. Regular Linux task are overhanging throughout the period of completion of real-time tasks by real-time kernel. Standard Linux kernel is treated as an inactive job by the synchronized programmer of the time stringent kernel, but when given a possibility to run the standard Linux kernel processes its individual programmer to program normal Linux task. The ordinary Linux tasks can at any instant be pre-empted by a synchronized task since the real-time kernel runs at a advanced priority.

Real Time kernel deals with another factor called Interrupt management. Real-time part gets and stores the interrupt when it is activated during the execution of a real-time task. The interrupt is given over to the standard Linux kernel when the real-time kernel is done. Real-time kernel executes the interrupts if there is a associated real-time handler for the interrupt. Otherwise stored interrupt is passed to the normal Linux if there are not any more real-time tasks to run. There were numerous different instruments used to pass the interrupts from real-time kernel to normal Linux kernel. Xenomai utilizes an Interrupt pipline from the ADEOS venture.

1.1 Real-Time Systems

Real time systems generally classified as hard real-time and soft real-time systems.

1.2 Hard Real-time Systems:

Strict planning prerequisites to meet due dates or unpredictable conduct can happen in Hard real-time systems. Aircraft auto-pilot framework is an example for hard real-time systems. The auto-pilot alters the course of the airplane promptly when it detects a potential impact with an adjacent question. Minimal latency during task switching, minimal jitter, run – to finish, need legacy pre-emptive multi-tasking and principally meet strict deadlines are some of the real prerequisites of hard real-time systems. Figure 1 shows the timing constraints in a hard real-time system using a time-utility function.

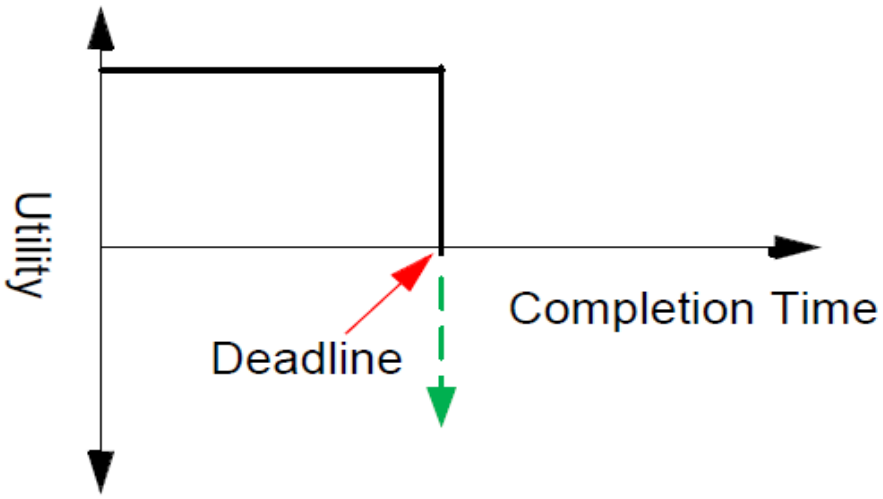


Figure 1. Hard Real-time System Time-utility Function

1.3 Soft Real-time Systems:

Meeting the deadlines is very important for soft real-time systems, But missing the same may not be catastrophic.CD player is an example for soft real-time system. There would be a delay in the actual time for few milliseconds to respond to the eject request. Timing constraints in a soft real-time system using a time utility function is shown in figure 2.

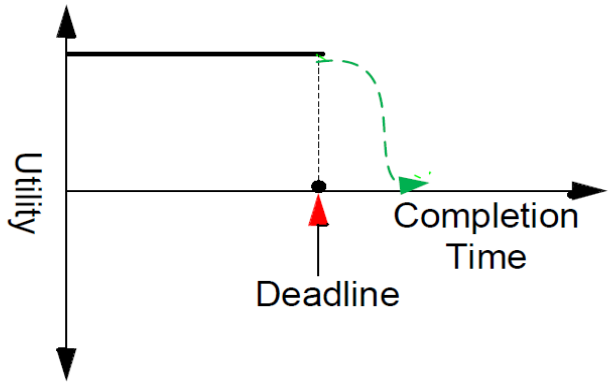


Figure: 1.2 Soft Real –time System Time-Utility Function

CHAPTER-2

LITERATURE SURVEY

1. Performance Comparison of VxWorks, Linux, RTAI and Xenomai in Hard Real-Time Application:

Authors: A. Barbalace, A. Luchetta , G. Manduchi, M. Moro

Abstract:

A cover set of execution estimations on execution of VME bus, MVME5500 sheets outfit with MPC7455 PowerPC processor which are running with four distinctive working frameworks: Wind River VX Works, Linux, RTAI and Xenomai. The Components of RTAI and Xenomai have ported focus to design. The Comparison of Framework of a specimen ongoing application with Interrupt latency, re-scheduling and between process communication times. The estimation of execution on Gigabit Ethernet organize correspondence was carried on target sheets.

Execution estimations demonstrated that tried open-source programming was reasonable for hard constant applications. The author has portrayed the achievement of new Linux conveyance taking into account of real structure of Xenomai and debian Linux.

Nevertheless Linux which has been successfully used for the real hard time control and beat nature of the trials. In this we have Porting Linux, ADEOS, RTAL, Xenomai to the MVME5500 Platform. We have built up the few patches of porting Linux, ADEOS, RTAL and Xenomai to the objective board.

In this we have the Performance Measurement , we focused on the two highlights intrude on inertness and rescheduling time. In this initially round for this, we concentrated on contrasts of interfere with inertness times of thought about frameworks. For this situation we will take program acloop first we introduce an Interrupt Service Routine. which straightforwardly peruses the ADC information enlist and writes to the DAC gadget. At that point for alternate frameworks which needs to code coordinated to a Linux module and it needs to been executed into a piece mode and the general measured postponement and to the circumstances required.

A first point has been the execution of Linux and in both the Jitter and delay and however the measure hold just for a framework and isn't stacked and it will diminish the workload will increment. The execution which are measured on alternate frameworks which are demonstrated to been just negligibly influenced by the workload and unless includes to a high interfere with rate. When the performance of Vxworks and of RTAI and it appear which that a exceptionally proficient scheduler for the local ongoing assignments. The last arrangement of tests we think about a system correspondence and for the situation we have the procedure sitting tight for the semaphore which does not compose at present being used the RFX-mod and in light of the detailed execution measures.

2. Implementation of Xenomai Framework in GNU/Linux Environment to run Applications in a Real Time Environment:

Authors: K.Sripath Roy, K.Gowthami

Abstract:

Point of the project is to portray the accomplishment of the new Linux movement of considered into Debian Linux and Xenomai Real-time structure. The strategies/Analysis is the affirmation is actuated through regarded necessity of ongoing working frameworks in late programming advancements. The Fundamentals destinations of movement such is propose exemplary working frameworks which is to consolidate is Xenomai base . It constant development is co-working with GNU/Linux working frameworks condition. It is general Xenomai innovation which goes for helping the application architects which is relying upon the customary RTOS to which expel to a GNU/Linux based execution condition, reworking their applications whole at no cost. The Porting of Xenomai is to Debian based Operating Systems and the assessment of execution parameters resembles clock test, Latency test on a General Purpose Computer

CHAPTER-3

THEORETICAL ANALYSIS

Xenomai:

An easy shift from normal real time operating system to GNU/LINUX will gives us a great approval to the real time embedded base giving emulators to support the regular RTOS application program interface is the strategy that can drive in a way which reduces the space between highly varied regular RTOS base to change to GNU/LINUX ,so that the application designers can depend on the regular RTOS to shift as easily to GNU/LINUX world. there is a need for common software framework for progressing work on emulators, so that operational equalities between the regular RTOS are prominent. The Xenomai technology focuses on bridging the lapse ,by giving uniform framework independent and generic emulation skin as the benefit of these comparability. it also owed to give an growing number of general RTOS emulators constructed above this layer. Xenomai believes the common characteristics and responses found between most embedded regular RTOS, significantly on thread scheduling and synchronization end points. These equalities are capitalize to construct a nucleus communicating a group of generic usage devices. These equalities are used in top stage interface that can be used in respective real time machine APIs

Xenomai deals with building real time operating system APIs accessible to machines working with Linux. When the Linux kernel can't catch the needs in case of timing parameters. Xenomai will supply and support for inflexible real time requirements

TO be abstract Xenomai can help in:

- Constructing, improving and running a time stringent system on traditional Linux.
- Shifting an application from proprietary RTOS to Linux base.
- Optimally working with RTOS tools alongside with regular and normal Linux applications.

To strengthen hard constant qualities to the Linux core, Xenomai utilizes a micro portion technique between the equipment assets and programming (Linux center bit). The miniaturized scale portion is responsible for working continuous works and interferes ,evading them from touching the Linux part ,so it can prevent the Linux machine from acquiring the hard constant micro-kernel internally,the Linux

bit works out of sight of this small scale bit as a less vital work that executes ordinary works, Xenomai gives hard time stringent change framework with ongoing maintenance for client space projects and practices the smaller scale piece in a stringent way. Likewise it gives different APIs to begin Real time assignments, counters, synchronizers and emulates diverse APIs known as SKINS for ongoing applications and errands alongside POSIX interface, RTAI, and so forth., they increment the simplicity to execute subsist RT application to Linux. Xenomai has ADEOS, time stringent nano kernel to direct ongoing limit execution.

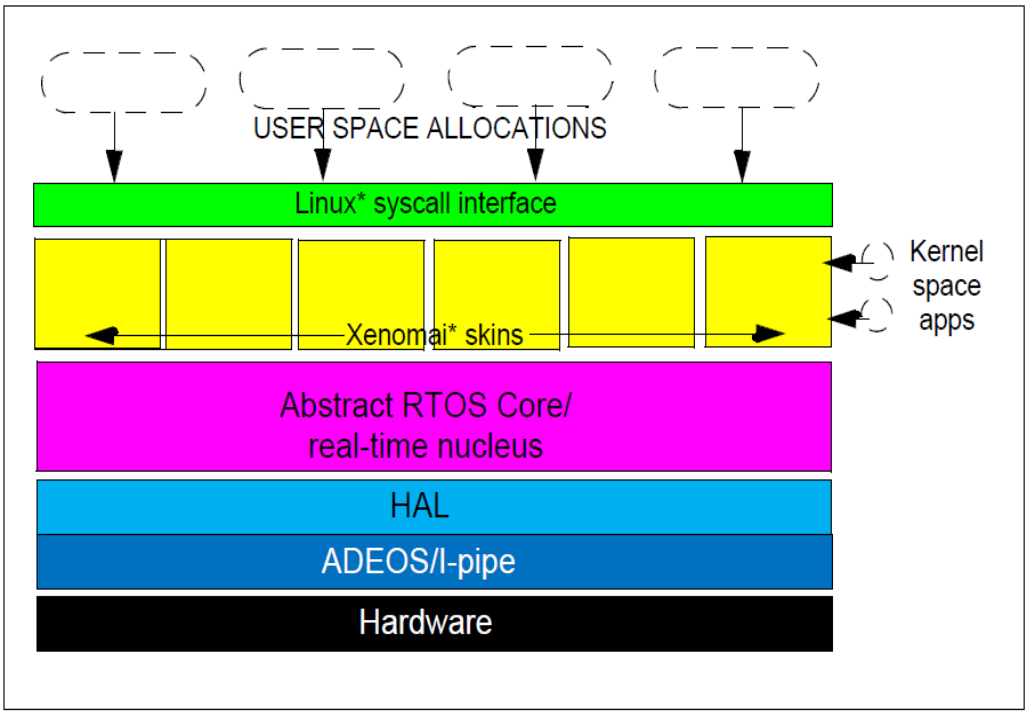


Fig-3.1 Xenomai Architecture

3.1 ADEOS pipelining:

Linux has a resource virtualization layer called ADEOS (adaptive domain environment for operating systems) in Linux kernel patch, which has a regular design proposed by Karim Yaghmour. It looks simple, if we go with current incarnation proposal. Even though it can work as a productive continuous framework empowering instrument by giving an approach to run an ordinary GNU/Linux condition and a RTOS, Side by agree with a similar equipment. ADEOS can enable various number of entities named domains to sustain one after the other on the same

hardware machine these entities named domains necessarily see each of them but all can see ADEOS in our case a domain is an operating system,but there is no premise is taken regarding experience of what is a domain. Even though all the domains will compete for processing interrupts or exceptions in order to complete a task.

All the domains will be given some priority values and are processed according to their priorities by straight forward virtualization capabilities .adeos can export a generic API to its client domain without the intervention of the CPU, which is a great advantage. As a result much of the interrupt traffic of client domains occurs at ADEOS level.

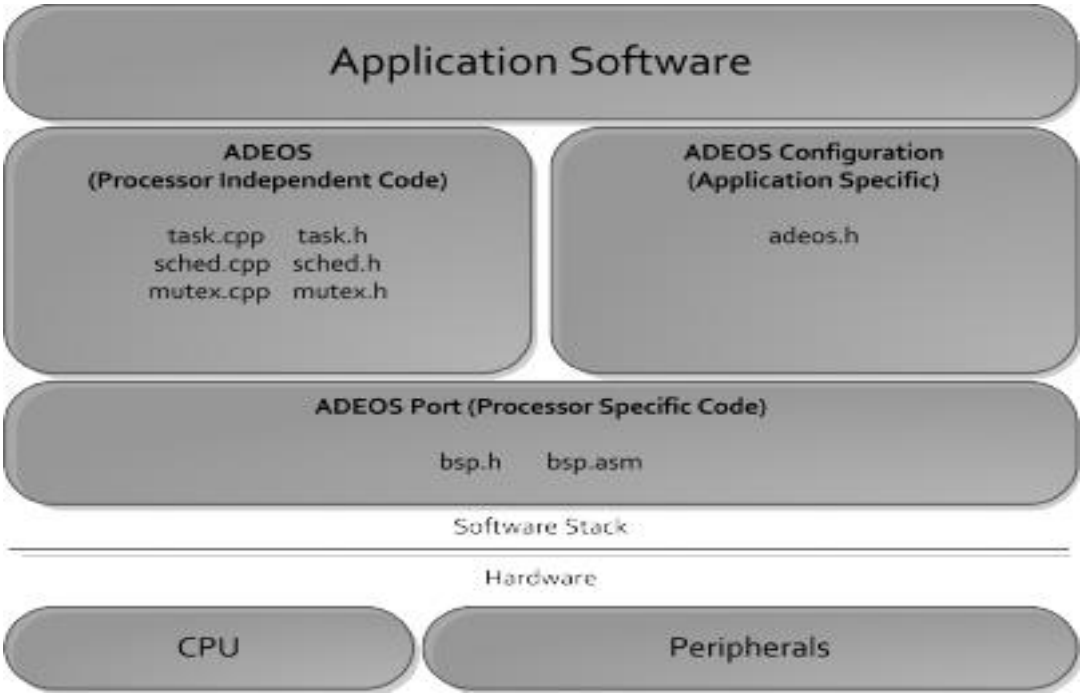


Fig-3.2 ADEOS pipelining

Lets suppose at an instance, a Linux system calls an issue from Xenomai thread working with the Xenomai entity will make the automatic shift of the caller thread to the Linux domain,even before the request is queued to normal Linux call handling kernel. On the other hand Xenomai thread which invokes a possibly blockage of Xenomai domain call will moved to the domain even before the interrupt is executed,so the caller will be in the control of real time kernel core.

This addition of both domains make Xenomai threads especially highly integrated into Linux environment. A common system invoking call path for Xenomai and

normal Linux applications makes the previous as a natural evolution process from the later. As a result, Xenomai threads supports both Linux signals semantics and tracing characteristic which enables the GDB native support to them.

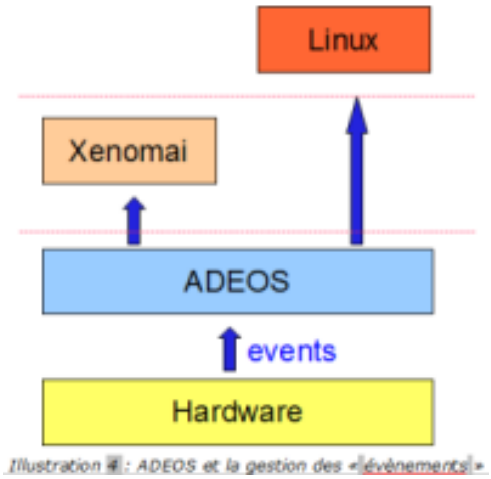


Fig-3.3 ADEOS with Xenomai

3.2 Features of Xenomai:

These are some of the features of the Xenomai-nucleus, and some key points are presented to make comparison with other real-time Linux distributions.

3.2.1.Multi-threading support

- 32 bit integer priorities
- per thread watchdogs
- support for priority inheritance
- pre-emptive scheduling algorithm
- 5-dimensional state model for processes
- Round-robin scheduling with same priority for threads

3.2.2 Basic synchronization support

- Support for forcible deletion with waiters awakening and for time-bounded wait
- Support for implementation shared with scheduler codes and priority inheritance

3.2.3 Timer and clock management

- A periodic mode and periodic mode
- Based on timer wheel algorithm
- Depending on timer operating mode nucleus transparently switches from periodic jiffies to time-stamp counter values
- Bounded worst-case time for stopping, starting and maintaining timers

3.2.4 Basic memory allocation

- With real time guarantees dynamic memory allocation (Real-time heap device)

3.3 Multi Processing

A thread of execution is the smallest arrangement of modified instructions that can be managed independently by a scheduler. Asymmetric Multi Processing involves multiple threads to be executed on multiple cores depending upon the configuration of the system.

At the operating system level, multiprocessing is referred as the execution of multiple simultaneous tasks in a system, with each task being performed on a separate CPU or core, as reverse to a single task at any one moment. Multi Threading is the ability of the processor or CPU to execute multiple threads or processes. It tries to enhance the utilization of a single processor by processing both threads and instruction sets in a parallel way. Multi Threading uses common memory heap for the allocation of the threads. Multi threading provides faster switching between the threads. Asymmetric multi-processing delegates system tasks to be executed by some processors and applications on others. This is basically not well organized as symmetric processing due to the reality that below specific characteristic one processor may be completely active while the other is inactive. At the operating system level, multiprocessing is rarely used to point out the implementation of various synchronous tasks in a system, with each task executing on a individual processor core, as conflicting to one task at any one instant. Multiprocessing reflects real parallel running of multiple processes using many processors. Multiprocessing certainly don’t convey one process or tasks uses many processors.

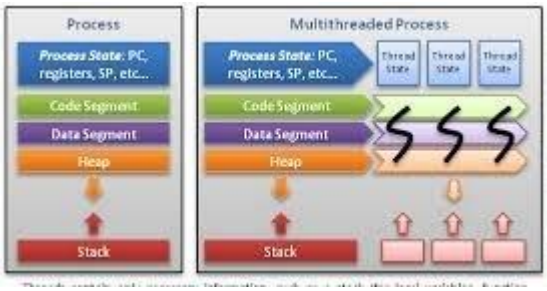


Fig-3.4 Multi Processing

There are two types of Multi Processing

1.Symmetric Multi Processing

2.Asyemmetric Multi Processing

Symmetric Multi Processing:

An SMP architecture referred as one where many similar cores connect to one another through a shared memory. Each core has equal access to the available memory resources

SMP systems are a set of identical cores working separately from each other. Each core, performing different instructions and processing various sets of data, has the capability to share common peripheral resources (memory, I/O device, interrupt system and so on) that are communicating using a system bus. each core has its own memory and can even access to shared memory with some different access rate.

The cores will begin executing the tasks from the normal ready queue. Every core may also have its own private queue of ready task to get executed. It must be taken care by the scheduler that no two processors execute the same process.

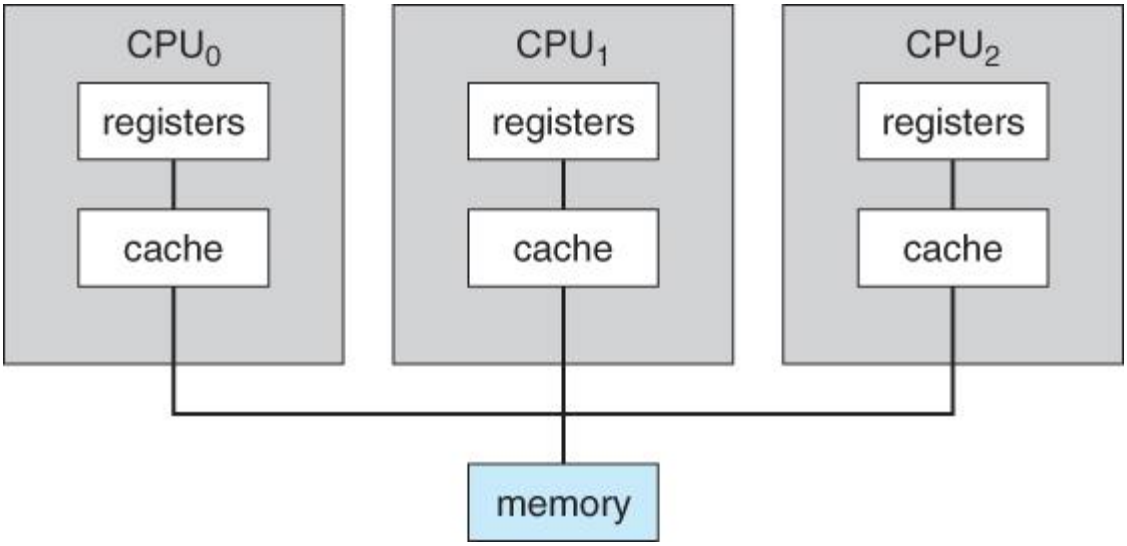


Fig-3.5 Symmetric Multi Processing

3.3.1 Asymmetric Multi Processing:

In an asymmetric multiprocessing system (AMP), all the cores are not treated equally. A system may or may not allow (either at the hardware or operating system level) only one CPU to execute operating system code or might allow only one CPU to perform I/O operations.

Asymmetric Multiprocessing has the master-slave relationship with the cores. here we have one master core that monitors the other slave core. The master core distributes tasks to slave core, or they may have some process to work upon.

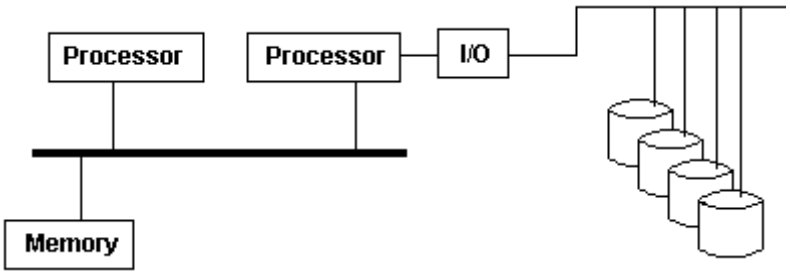


Fig-3.6 Asymmetric Multi Processing.

3.3.2 Mutex (Mutual Exclusion)

It is a way of serializing access to shared resources and don't want thread to be modified another thread. It allows you to programmer to create a shared data or resource to protocol for serializing access. It is a logical a mutex a lock that can virtually attach to some resource. If thread wishes to change or read value from a common resource, thread must first gain the lock and it has a lock it does it wants a shared resource without remaining threads accessing the common resources**.**

3.4 Scheduling

A schedule or a timetable, as a fundamental time-administration apparatus, comprises of a list of times at which possible tasks, events, or actions are proposed to make put, or of an arrangement of events in the sequential request in which such things are expected to occur. The way toward making a schedule - choosing how to arrange these tasks and how to commit assets between the variety of conceivable tasks is called scheduling.

3.4.1 Scheduling Criteria

There are many parameters to look upon when choosing to the "best" scheduling algorithm for an instance and environment, including:

CPU usage - Ideally the CPU would be occupied 100% of the time, in order to squander 0 CPU cycles. On a genuine framework CPU use should run from 40% (softly stacked) to 90% (intensely stacked).

Throughput - Number of tasks finished per unit time. May extend from 10/second to 1/hour relying upon the particular procedures.

Turnaround time - Time required for a specific task to finish, from starting time to completion of task (Wall clock time).

Waiting time – The amount of time forms spend in the prepared line hanging to get on the CPU.

(Average Load – The normal number of tasks sitting in the prepared line holding up to get into the CPU. Revealed in 1-minute, 5-minutes, and 15-minutes midpoints by "uptime" and "who").

Response time - The time taken in an intelligent program from the issuance of a charge to the initiate of a reaction to that order.

At the point when all is said in completed one needs to enhance the typical estimation of a criteria (Maximize CPU utilize and throughput, and farthest point all the others). However few times one needs to achieve something unique, for instance, to restrict the most extraordinary Sometimes it is most alluring to limit the difference of a criteria than the genuine esteem. I.e. clients are more tolerating of a steady unsurprising framework than a conflicting one, regardless of whether it is a smidgen slower.

3.4.2 First-Come First-Serve Scheduling, FCFS

- Unfortunately, FCFS can result in some very large average wait times, particularly if the first task takes to get there a large period.
- FCFS is extremely straightforward - Just a FIFO line, similar to clients holding up in line at the bank or the mail station or at a duplicating machine.

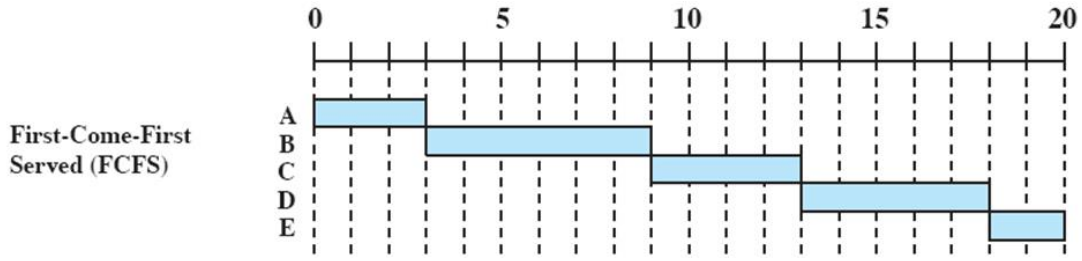


Fig-3.7 First-Come First-Serve Scheduling

3.4.3 Shortest-Job-First Scheduling, SJF

- The point with SJF technique is to pick the speediest quickest little task that takes to be done, complete it and expel off the beaten path in the first place, and afterward pick the following littlest speediest errand to work straightaway.
- Technically this calculation picks an assignment basing on the following most limited CPU burst, not the general errand time.
- SJF can be turned out to be the speediest booking technique, however it experiences one imperative issue: How do the developer knows what amount of time the following CPU burst will be?

- For longer bunch employments this should be possible basing upon the limits that clients preset for their assignments when they submit errands, which urges them to set low cutoff points, however chances their having to re-enter the undertakings on the off chance that they set the breaking point too low. However that does not work for here and now CPU planning on an instinctive system.

- Another decision is measurably measure the run time characteristics of occupations, particularly if comparable errands are keep running on and on and normally. In any case, before long that genuinely isn't a plausible decision for without a moment's hesitation CPU getting ready for this present reality.

- most practical method is prediction of the length of the next burst, basing on some previous measurement of recent burst times for this task.

Fig-3.8 Shortest Job First Scheduling

3.4.4 Priority Scheduling

- Priority arranging is a more wide example of SJF, in which each task is alloted a need and the undertaking with the most urgency gets booked first. SJF uses the opposite of the accompanying expected burst time as its need - The more diminutive the typical burst, the higher the need.

- By practice we found that, needs are executed utilizing whole numbers inside a settled range, however there is no settled upon tradition with respect to whether "high" urgencies utilize extensive numbers or little numbers. This method utilizes low number for urgency values, with 0 being the most elevated conceivable need.

- Priorities can be given out either inside or remotely. Interior needs are consigned by the OS using criteria, for instance, ordinary burst time, extent of CPU to I/O development, structure resource use, and diverse segments open to the piece. External needs are doled out by customers, in light of the importance of the movement, costs paid, authoritative issues, etc.

- Priority planning can be either preemptive or non-preemptive. Priority arranging can encounter the evil impacts of an important issue known as indeterminate blocking, or starvation, in which a low-require task can hold up ceaselessly in light of the way that there are continually some unique occupations around that have higher need.

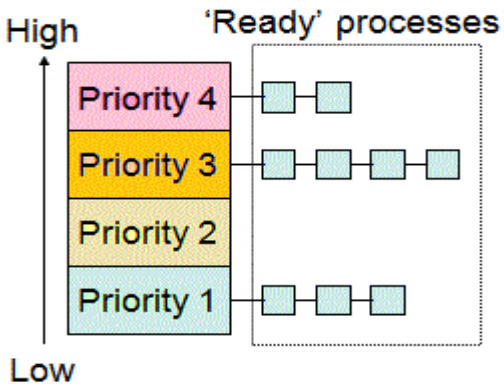


Fig-3.9 Priority Scheduling

3.4.5 Round Robin Scheduling:

- Round robin strategy is like FCFS scheduling, aside from that CPU bursts will be set with limits called time quantum.

- When an task is given the CPU, a clock is set for the value that is already stored for a period quantum.

- If the task is finished burst before the time quantum clock lapses, at that point it is swapped out of the CPU simply like the typical FCFS algorithm.

- If the timer goes beyond the quantum, then the task is changed from the CPU and will be placed at the end of ready queue

- The circular queue is chosen for prepared line, so when all tasks have had a turn, at that point the scheduler permits the primary assignment another possibility, etc.

- RR methodology can give the impact of all tasks are sharing the CPU equally, even though the normal hold up time can be bigger than with other scheduling approach.

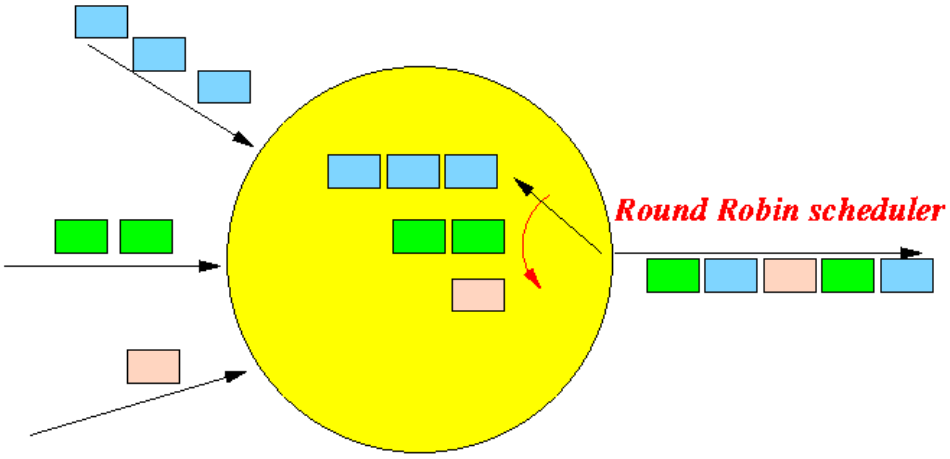


Fig-3.10 Round Robin Scheduling

3.4.6 Multilevel Queue Scheduling

- When tasks can be promptly classified, at that point numerous different queues can be built up, each scheduling whatever scheduling algorithm is most proper for that kind of task, as well as with various parametric changes.

- Scheduling must in like manner be done between queues, that is scheduling one queue to get time in regard to various queues. Two essential decisions are strict need (no urgency in a lower need line keeps running until the point when all high priority queues are free) and round-robin (each line gets a period , perhaps of various sizes).

- Note that under this method tasks can't be changed from queue to queue - Once they entered into a queue, that is their queue until they complete the specific task.

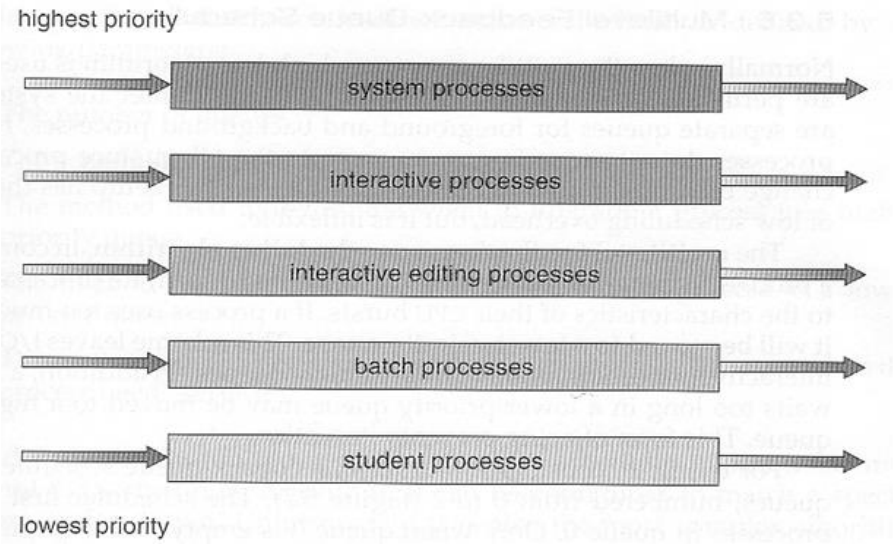


Fig- 3.11 Multilevel Queue Scheduling

3.4.7 Thread Scheduling

- Only the kernel threads are scheduled by job scheduler.

The thread library links the user tasks to the kernel threads by thread libraries- The OS (and in particular the scheduler) is unaware of them.

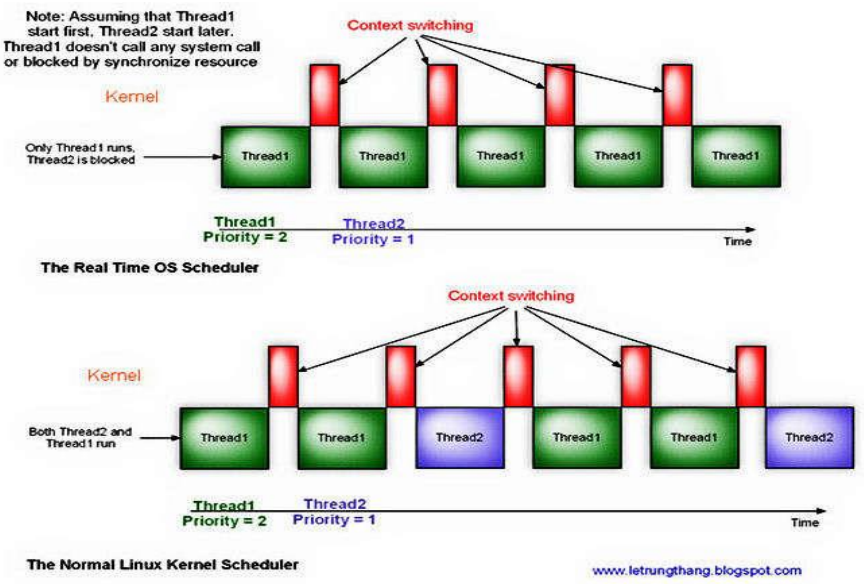


Fig-3.12 Thread Scheduling

3.5 Contention Scope

- The rate in which threads compete for accessing of peripheral resources is known as contention scope.
- On machines executing many-to-one and many-to-numerous tasks, Process Contention Scope, PCS, is come about, in light of the fact that opposition happens among the threads that are executing a similar task (This is the administration/scheduling of different client threads on a single kernel thread, and is overseen by the thread library.)
- System Contention Scope, SCS, requires the system scheduler scheduling kernel threads to execute on one or numerous CPU's. Systems tailing one-to-one task (XP, Solaris 9, Linux), will only utilize SCS.
- PCS scheduling is normally done with urgency, where the programmer can program and also modify the priority value of threads instantiated by his or her programs. Even among threads having same priority time slicing is not guaranteed.
- In Linux, threads (likewise called Lightweight Processes (LWP)) began inside a program will have "thread group ID" and the program's PID. Each thread will have its own particular thread ID (TID). To the Linux kernel's scheduler, threads are standard procedures which will work with sharing of specific peripherals to finish task.

3.6 Multi Threading:

Multi threading is the capacity of a program or a operating system procedure to deal with its utilization by in excess of one user at any given moment and to try and deal with various demands by a similar user without having numerous copies of the programming running in the PC. Every user ask for a program or system service (and here a user can likewise be another program) is monitored as a thread with a different identity. As projects work on behalf of initial request for that thread and are interrupted by different requests, the status of work on behalf of that thread is monitored until the point when the work is finished.

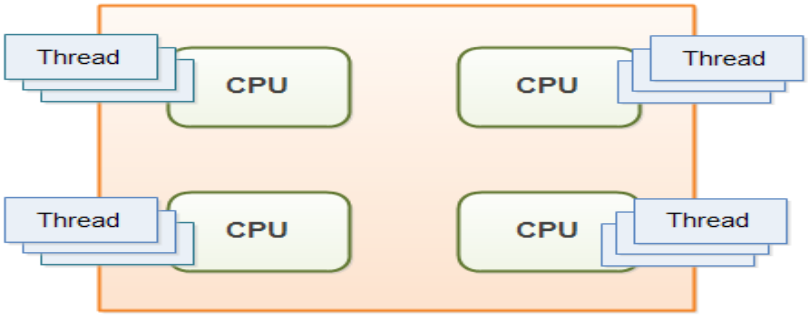


Fig-3.13 Multi Threading

Viewing threads in a CPU:

In ps command, "-T" option enables thread views. The following command list all threads created by a process with <pid>.

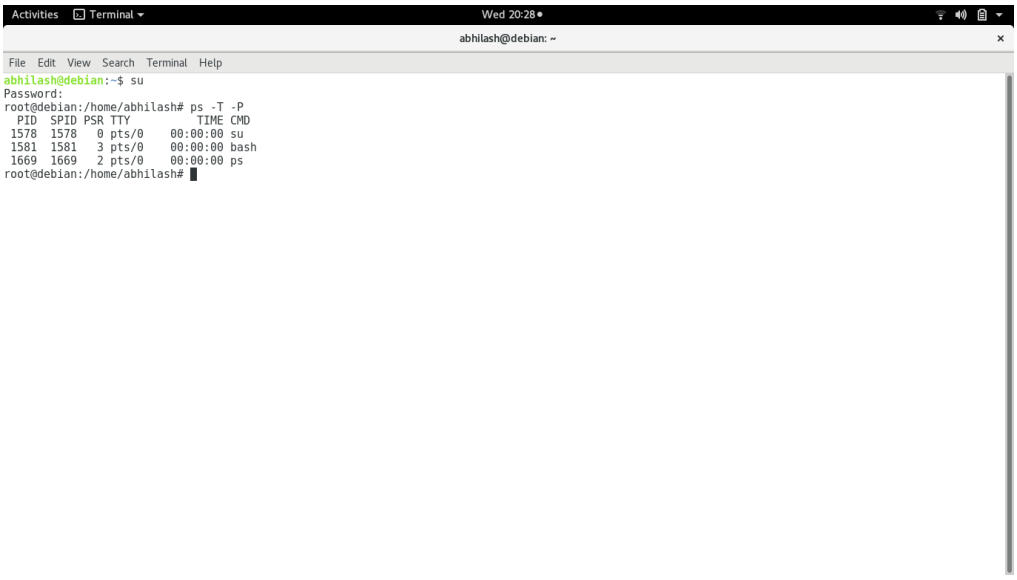


Fig :3.14 Viewing of threads in CPU

The top command can show a real-time view of individual threads. To enable thread views in the top output, invoke top with "-H" option. This will list all Linux threads

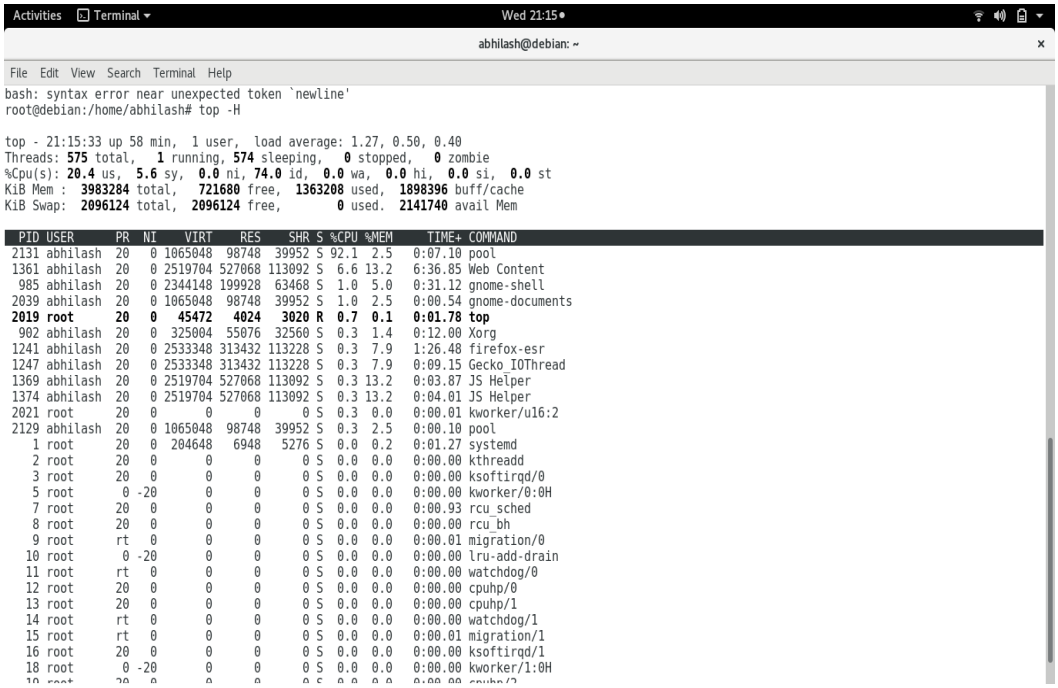


Fig:3.15 Viewing threads per task in CPU

an user-friendly approach to view threads per task is using htop, an ncurses-based interactive task viewer. This program allows the user to monitor individual threads in tree views.

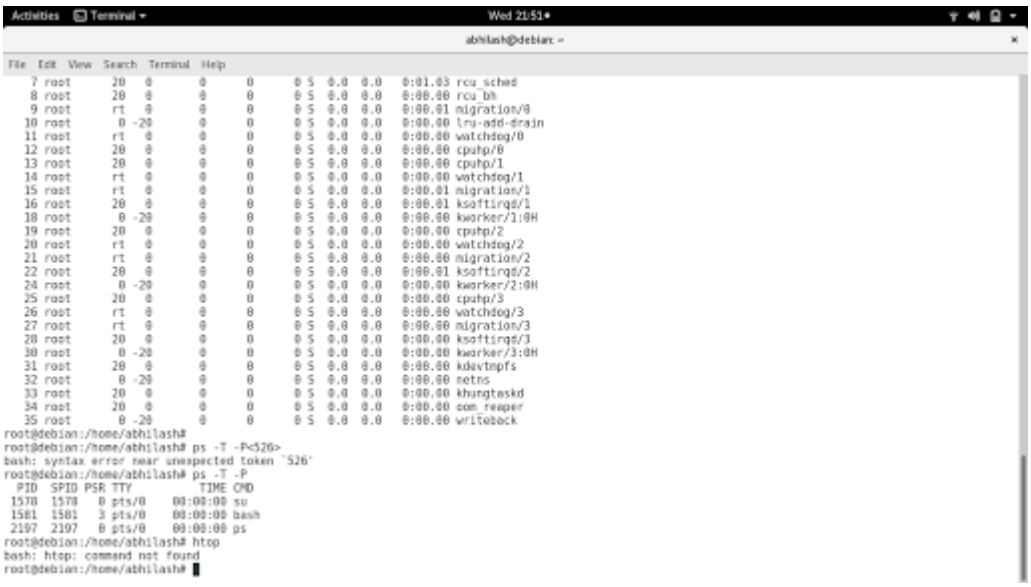


Fig-3.16 Viewing Threads in CPU using htop command

Pthread Scheduling

- The scope contention to specify is provided by Pthread library:

- PTHREAD_SCOPE_PROCESS manages the tasks by operating PCS, by scheduling user threads to accessible Light Weight Proceesess using the many-to-many approach.
- PTHREAD_SCOPE_SYSTEM manages threads deploying SCS, by user threads to particular Light Weight Proceeses, efficiently working with a one-to-one model.
- “getscope” and “setscope” are the methods which are used finding and entering the scope contention

CHAPTER-4

EXPERIMENTAL INVESTIGATION

4.1 Installation steps:

The installation process for building the kernel is as follows

- Download stable Linux version
- Download Xenomai-3.0.5 version
- Collect stable I-pipe version
- Build essential Linux packages
- Patch Linux Kernel with Xenomai
- Configure Linux Patch
- Compile Kernel
- Testing the Kernel

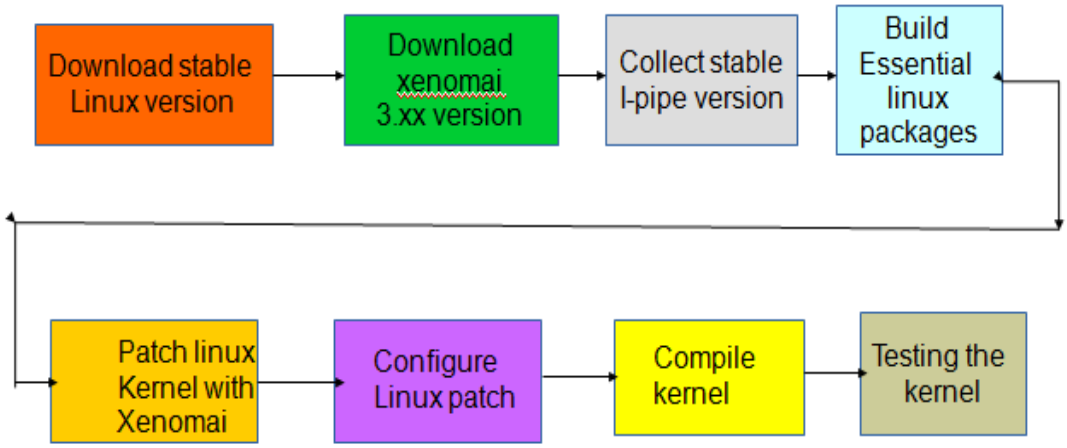


Fig.4.1 Installation flow of Xenomai

4.1.1 Downloading stable Linux source:

Stable Linux Kernel Linux i.e Linux-4.9.51 has to be downloaded from the official website

4.1.2 Downloading Xenomai 3.0.5 version:

Xenomai-3.0.5 version has to be downloaded

4.1.3 Collecting Stable I-pipe Core:

Download the stable I-pipe core-4.9.51-x86-4 patch

4.1.4 Build essential packages in Linux:

Essential packages were built in downloaded Linux kernel using

"apt-get install build-essential gcc libncurses5-dev libssl-dev" libncurses5-dev package will make the kernel easier to configure.

4.1.5 Patch Linux Kernel with Xenomai

Now we need to patch our downloaded stable Linux kernel with xenomai using

"/scripts/prepare-kernel.sh-Linux=../../Linux-4.9.51/Linux-4.9.51/--ipipe=../../ipipe-core-4.9.51-x86-4.patch-arch=x86-4"

4.1.6 Configuring Patched Kernel:

Using **"make localmodconfig"** detects currently running kernel components and marks them for compilation.

"make menuconfig" is used to choose what to compile. Here we make changes and generate a .config file.

During configuring we need to enable and disable some settings

4.1.7 Power Management:

Power management should not be disabled globally, the only which are to be disabled in this area are

- **CONFIG_APM**
- **CONFIG_ACPI_PROCESSOR**

CPU Frequency scaling:

- **CONFIG_CPU_FREQ – Disable**
  Numerous CPUs change the TSC checking recurrence which makes it pointless for exact planning when the CPU clock can change. Also a few CPUs take a few milliseconds to increase to full speed. Thus, by incapacitating this enables the CPU to be adjusted with workload.
- **CONFIG_CPU_IDLE – Disable**
  This enables the CPU to enter profound rest states , expanding the time it removes to get from these rest states. Timers used by Xenomai stop functioning when entered into these sleep states.

- **CONFIG_APM – Disable**
  Power management control to the BIOS is assigned by the APM model. BIOS code is never composed in light of best inertness. APM schedules are conjured with SMI need if designed, this sidesteps the I-pipe totally.

- **CONFIG_ACPI_PROCESSOR – Disable**

  The ACPI processor module disables the local ACPI which will cause the Xenomai timer initialization to fail due to this reason we disable this option.

- **CONFIG_INTEL_IDLE – Disale**

  This causes huge latencies due to the fact that the ACPI clock that Xenomai uses may not fire any longer, likewise simply like CONFIG_ACPI_PROCESSOR, this sit out of gear driver sends the CPU into deep C states.

4.1.8 Compile Kernel:

Now we need to compile our Linux kernel using

- **"Sudo make-j$(nproc—all)"**

  This command is used to compile the kernel. "$" represents number of processors
- "**make modules_install"**

  This command is used to finish compiling the rest of the kernel
- "**make install"**

  This will automatically copy the kernel to your / boot folder and generate the appropriate files to make it work.

4.2 Test Cases:

Before Performing the functional testing on the kernel we need to test whether the Xenomai features are patched with our compiled Linux kernel or not, which can be done using the commands "**dmesg grep | xenomai", "dmesg grep | xenomai-i".**
Many benchmarking tools for both real-time and user-space kernels comes with Xenomai and they are accessible from /**usr/xenomai/bin.** The tests include the following:

- Clock-test
- Switch-bench
- Latency

4.2.1 Clock Test:

Clock test is a bit of the Xenomai test which tests the Clock. For each CPU, it prints again and again a time offset (contrasted and the reference gettimeofday()), the amount of wraps and greatest wrap in microseconds and a drift value.

For this program to run you need to run **./clocktest** in the suitable Xenomai enabled kernel

Options:

Clock test accepts the following options:

-C<clock_id>

Clock to be tested, default=0 (CLOCK_REALTIME=0, CLOCK_MONOTONIC=1, CLOCK_HOST_REALTIME=42)

-T<test _duration_seconds>

default=0 (Never stop, ^C to end)

-D

Extra diagnostics for CLOCK_HOST_REALTIME will be printed

4.2.2 Latency:

This test is the best indicator of your real-time performance, it verifies the overall performance of your system.In the event that you have effectively introduced an accurately working continuous part this test will tell you quickly. It gauges the distinction in time between the time when an task is really called by the scheduler and the time between the normal switch time. This test prints minimum, average , and maximum latencies for that period and in addition minimum and maximum overall latencies that happened over the whole test for each one moment. Open up some different projects, duplicate a few records from one area to other , and load the system association with perceive the amount it influences the idleness. You should discover somewhat lesser latencies with part space test than client space test.

The execution of the continuous undertakings is constrained by the most extreme inertness. At the point when RTAI plays out an adjustment at startup that tries to limit the jitter in the ongoing errand and foresees the call we get a negative time in the dormancy test. You additionally ought not perceive any invades, which happens when the inertness totally surpasses your nominal period.

We get a consequence of most extreme latency of a few hundred microseconds and you may have a SMI (System Maintenance Interrupt) issue, when you occasionally observe an overrun (maybe at regular intervals). This element can be found on certain chipsets e.g. Intel 82845. A few PCs may harm due to overheat on the disabling of SMI. Other "latency killers" are: utilizing a quickened X-server, USB inheritance bolster, heavy DMA exercises (utilizing the hard plate), control administration (APM and ACPI), and CPU recurrence scaling. On the disabling all of these in the kernel already, check your BIOS and check whether you can disabled them there.

Options:

Latency accepts the following options:

-h

Histograms of max, avg, min latencies will be printed.

-g<file>

Dump histogram to <file> in a format which is easily readable with gnu plot. In scripts/histo.gp in Xenomai sources distributions u can find an example script for gnu-plot.

-s

Statistics of max, avg, min latencies will be printed.

-H<histogram-size>

Default=200, increase if your last bucket is full

-B<bucket-size>

Default =1000ns, decrease for more resolution

-p<period_us>

Sampling period

-I<data-lines per header>

Default=21, 0 to suppress headers

-T<test_duration_seconds>

Default=0, so ^C to end

-q

Suppresses RTH, RTD lines if –T is used

-D<testing_device_no>

Number of testing devices, default=0

-t<test_mode>

0=user task (default), 1=kernel task, 2=timer IRQ

-f

For each new max latency freezes trace

-c<cpu>

Pin measuring task down to given CPU

-P<priority>

Task priority (test mode 0 and 1 only)

-b

Break upon mode switch

4.2 3 Switch test:

Thread context switches can be tested using this test. Switch test creates threads of various types and prints the count of context switches every second and attempts to switch context between these threads.

The data with respect to the most extreme measure of time RTAI needs to disable the interrupts is given by this test. The test utilizes a repeated grouping of resume/suspend/and semaphore flag/hold up calls under a substantial handling load. The exchanging time ought to be not as much as the most extreme inertness time. The characteristic disadvantage of utilizing universally useful CPU for constant applications, the genuine inertness impediment is at times because of RTAI .

Options:

The characteristics of a thread to be created is specified by each threadspec

threadspec= (rtk|rtup|rtus|rtuo)(_fp|_ufpp|_ufps)*[0-9]*

rtus-running in secondary mode the user-space real-time thread

rtk-kernel space real-time thread

rtuo- oscillating between primary and secondary mode for a user-space real-time thread

rtup-running in primary mode the user-space real-time thread

_fp—the thread created will have the XNFPU bit armed, this is valid only for rtk

_ufps—with the secondary mode the created thread will use the FPU, this is invalid for rtk and rtup

_ufpp—with the primary mode the created thread will use the FPU, this is invalid for rtus

[0-9]—ID of the CPU where the created thread will run is specified by this, if it is not specified it is 0 by default

Switchtest accepts the following options:

--help, -h

Usage information will be printed and exit

--lines<lines>,-I<lines>

Headers every<lines>lines will be printed

--quiet or –q

Printing every second the count of ncontext switches in the program will be prevented

--timeout<duration>,-T<duration>

Test duration will be limited to <duration>seconds

--nofpu, -n

Any utilization FPU instructions will be shown.

4.3 Implementing Asymmetric Multiprocessing :

In order to implement Asymmetric processing first we need set few pre-requisites.

- Preferred Platform
- Selection of CPU
- Kinds of tests to be performed

4.3.1 Preferred Platform:

We have chosen Linux to be the suitable platform because of its Free and Open Source Nature having a larger community working towards solving problems. Support of the community and availability of API can be used to solve real time problems. Hence we implemented asymmetric processing on Linux kernel

Linux is a Unix-like working framework that was intended to give PC clients a free or ease working framework

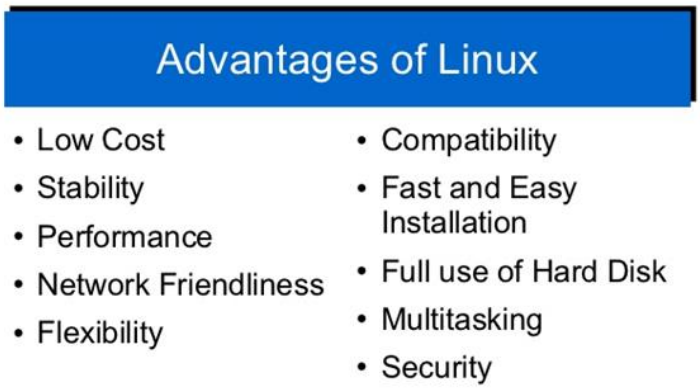


Fig:4.2 Advantages of Linux

Low cost:

When using a Linux System you don't need to contribute vitality and money to get licenses since Linux and a lot of its item go with the GNU General Public License.

Stability:

Linux doesn't should be rebooted intermittently to keep up execution levels. It doesn't solidify up or back off after some time because of memory leaks and such.

Performance:

Linux gives eager prevalent on workstations and on frameworks. It can manage shockingly immense amounts of customers in the meantime.

Network friendliness:

Linux has solid help for arrange usefulness; customer and server frameworks can be effortlessly set up on any PC running Linux. It can perform tasks, for example, network backups faster and more dependably than elective frameworks.

Flexibility:

Linux can be used for world class server applications, work region applications, and introduced systems. You can spare circle space by simply presenting the sections required for a particular use. You can constrain the usage of specific PCs by introducing for instance just chose office applications as opposed to the whole suite

Compatibility:

It runs all normal Unix programming bundles and can process all regular file formats.

Choice:

The extensive number of Linux conveyances gives you a decision. The center functionalities are the same most programming keeps running on generally circulations.

Fast and straightforward foundation Most Linux assignments go with straightforward foundation and setup programs. Noticeable Linux scatterings go with instruments that make foundation of additional programming to a great degree straightforward as well.

Multitasking:

Linux is designed to do many things at the same time.

e.g., a large printing job in the background won't slow down your other work.

Security:

Linux is a standout amongst the most secure working frameworks. "Walls" and adaptable document get to authorization frameworks avert access by undesirable guests or infections. Linux clients need to alternative to choose and securely download programming, free of charge, from online archives containing a large number of high quality packages.

4.3.2 Selection of CPU:

We have implemented asymmetric multi processing on various multi core processors like a. x86 i3 processor with Frequency- 1900 MHz. b. AMD II X3 720 Processor, Frequency- 800 MHz and c. x8 i5 Processor, Frequency- 1400.158 MHz. The system performance and latency was measure for the same. We chose general purpose computing platforms as a base to test the computational power as we are testing the system to work as a IoT Gateway Server.

4.3.3 Performing tests :

At first we have created many threads using POSIX APIs and set priority the tasks .Later by using scheduler we assigned the tasks to the specific CPU cores.

For example if there are 10 tasks , and if tasks 2, 7 are assigned to CPU1 and 1,4,8,9,10 are assigned to CPU3 and tasks 3,5,6 are assigned to CPU4,Here CPU2 is left idle. Then the CPU 1,3,4 perform only related and internal tasks where as CPU2 performs only internal tasks, rest of the time it remains idle. Hence in this way we have implemented asymmetric processing on multi core and checked for CPU performance and measured the start time, end time and latency of all the tasks.

4.4 Methodology:

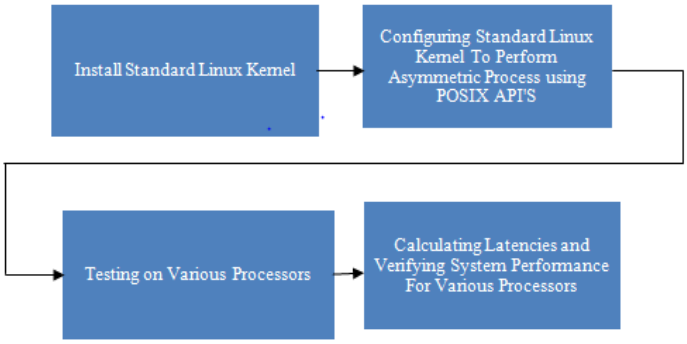


Fig:4.3 Block diagram and explanation

4.4.1. Installing Standard Linux Kernel.

The first step is to install standard Linux kernel and configuring the kernel to perform Asymmetric Multi Processing.

4.4.2 Configuring standard Linux kernel for Asymmetric multi processing using POSIX API'S.

After installing Standard Linux kernel we implemented multi processing with the help of POSIX (portable operating system interface) threads.

We implement multi processing by creating and processing large number of POSIX threads .As a part of asynchronous multi processing, we allocated specific threads to a particular processor. We defined 4 tasks and tasks were assigned to each processor

4.4.3 Testing on various processors

Next we worked on various processors to implement asynchronous process by creating and processing a number of threads where threads were assigned to process specific tasks. We implemented multi processing successfully and thread parameters for each thread are obtained.

4.4.4. Calculating Latencies and verifying system performance for various processors.

After getting the thread related parameters by asynchronous processing, we found latency for each thread. It gives timing for each thread, which is most important for an real time operating system. We also verified multiprocessing on the system level using system monitor performance. Later the graphs were plotted for asynchronous processing for various processors.

4.5 Test for performance:

After the successful building of the kernel, we need to test the behavior and performance of the kernel to justify the real time behavior of the kernel. To verify the functional behavior and performance metrics of the kernel, we need to carry some standard test benches like latency test.

CHAPTER-5

SOURCE CODE

5. 1 Asynchronous Multi Processing.C(for implementation of A.M.P)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
#define QUEUE_SIZE    2000
#define TOTAL_THREADS 500
#define ITERATIONS    1
clock_t start[TOTAL_THREADS];
clock_t end[TOTAL_THREADS];
double calctime[TOTAL_THREADS];
double lat[TOTAL_THREADS];
int s, j;
cpu_set_t cpuset;
pthread_t workers[TOTAL_THREADS];
typedef pthread_barrier_t *waiting_thread;

//struct to simulate fifo-based mutex-----------------------------------------
typedef struct
{
        int  init;
        waiting_thread    threads[QUEUE_SIZE];
        pthread_mutex_t   mutex, change_mutex;
```

```
        pthread_cond_t    queue_change;
        pthread_barrier_t queuing_sync;
        unsigned int      front, count, max;
} thread_queue;
static thread_queue main_queue = { 0 };

//END struct to simulate fifo-based mutex-------------------------------------
//functions for dealing with queue-------------------------------------------
//initialize the queue
static int init_main_queue()
{
        if (main_queue.init) return 1;
        memset(main_queue.threads, 0, sizeof(waiting_thread) * QUEUE_SIZE);
        if (pthread_mutex_init(&main_queue.mutex, NULL) != 0)           return 0;
        if (pthread_mutex_init(&main_queue.change_mutex, NULL) != 0)    return
0;
        if (pthread_cond_init(&main_queue.queue_change, NULL) != 0)     return 0;
        if (pthread_barrier_init(&main_queue.queuing_sync, NULL, 2) != 0) return 0;
        main_queue.front = 0;
        main_queue.count = 0;
        main_queue.max   = QUEUE_SIZE;
        main_queue.init  = 1;
        return 1;
}
//clean up the queue
static int fini_main_queue()
{
```

```
	//(there should probably be some sort of 'main_queue.threads' cleanup)

	if (!main_queue.init) return 1;
	pthread_mutex_destroy(&main_queue.mutex);
	pthread_mutex_destroy(&main_queue.change_mutex);
	pthread_cond_destroy(&main_queue.queue_change);
	pthread_barrier_destroy(&main_queue.queuing_sync);
	return 1;
}

//push a thread onto the queue (from the thread itself)
static int push_waiting_thread(waiting_thread tThread)
{
	//obtain a queue lock
	if (!main_queue.init || pthread_mutex_lock(&main_queue.mutex) != 0)
	return 0;
	if (main_queue.count == main_queue.max)
	{
	pthread_mutex_unlock(&main_queue.mutex);
	return 0;
	}

	//notify the queuing thread of a change; the lock on the mutex will
	//prevent it from continuing until the change is made
	pthread_cond_broadcast(&main_queue.queue_change);

	//add the thread to the queue
```

```
	main_queue.threads[(main_queue.front + main_queue.count++) %
main_queue.max] = tThread;
	pthread_mutex_unlock(&main_queue.mutex);

	return 1;
}

//wait for access to the "resource" (from the thread needing it)
static int wait_for_resource(waiting_thread tThread, int iIdentity)
{
	//add the thread to the queue and wait for its number to come up
	if (!push_waiting_thread(tThread)) return 0;
	fprintf(stderr, "thread %i waiting\n", iIdentity);
	int outcome = pthread_barrier_wait(tThread);
	return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}

//"unlock" the simulated mutex when finished with it
static int finish_with_resource()
{
	if (!main_queue.init) return 0;
	int outcome = pthread_barrier_wait(&main_queue.queuing_sync);
	return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}

//call the next thread in the queue (from the queuing thread)
static int pop_waiting_thread()
```

```
{
	int outcome;

	//obtain a queue lock
	if (!main_queue.init || pthread_mutex_lock(&main_queue.mutex) != 0) return
0;

	//if the queue is empty, block until a thread is added
	if (!main_queue.count)
	{
	pthread_mutex_unlock(&main_queue.mutex);
	//(there is a chance 'pthread_cond_broadcast' will happen between the
	//line above and the line below; therefore, a 'nanosleep' might be in
	//order here. a 'pthread_barrier_wait' wouldn't be any easier.)
	pthread_mutex_lock(&main_queue.change_mutex);
	pthread_cond_wait(&main_queue.queue_change,
&main_queue.change_mutex);
	pthread_mutex_unlock(&main_queue.change_mutex);
	return 0;
	}

	waiting_thread next = main_queue.threads[main_queue.front++];
	--main_queue.count;
	main_queue.front %= main_queue.max;
	pthread_mutex_unlock(&main_queue.mutex);

	//continue the thread
	//(this will cause a segfault if 'next' has been destroyed already)
```

```
        outcome = pthread_barrier_wait(next);
        if (outcome && outcome != PTHREAD_BARRIER_SERIAL_THREAD)
return 0;


        //wait for the thread to finish with the resource
        outcome = pthread_barrier_wait(&main_queue.queuing_sync);
        return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}


//END functions for dealing with queue----------------------------------------
//threads---------------------------------------------------------------------
//queuing thread
static void *queuing_thread(void *iIgnore)
{
        struct timespec delay = { 0, 100 * 1000 * 1000 };


        if (!main_queue.init) return NULL;
        fprintf(stderr, "queuing thread started\n");
        while (1)
        {
        pop_waiting_thread();
        nanosleep(&delay, NULL);
        }
        fprintf(stderr, "queuing thread finished\n");
        return NULL;
}
void perform_add(){
```

```
int a=1000,b=2000;
int sum=a+b;
}
void perform_mul(){
int a=10,b=20;
int prod=a*b;
}
void perform_pi_calc(){
int i;
double p = 0;
for ( i = 1; i <= 10000; i++) {
 if ( i % 2 == 0) {
 p = p - (4.0 / (2 * i - 1));
}
else {
 p = p + (4.0 / (2 * i - 1));
}
//printf( "%f\t", p );
}

}
//worker threads
static void *worker_thread(void *iIdentity)
{
        int identity = *(int*) iIdentity, iterations = ITERATIONS;
        clock_t temp,st,en;
        if(lat[identity]==0){
```

```
temp=clock()-start[identity];
lat[identity]=((double)temp)/CLOCKS_PER_SEC;
}
fprintf(stderr, "thread %i started\n", identity);
pthread_barrier_t barrier;
if (pthread_barrier_init(&barrier, NULL, 2) != 0) return NULL;
while (iterations-- > 0 && wait_for_resource(&barrier, identity))
{
s = pthread_getaffinity_np(workers[identity], sizeof(cpu_set_t), &cpuset);
for (j = 0; j < CPU_SETSIZE; j++)
  if (CPU_ISSET(j, &cpuset)){
     printf("THREAD %d USING CPU %d\n",identity,j);
                if(j==0){
                        st=clock();
                        perform_add();
                        en=clock()-st;
                }
                else if(j==1){
                        st=clock();
                        perform_mul();
                        en=clock()-st;
                }
                else{
                        st=clock();
                        perform_pi_calc();
                        en=clock()-st;
                }
```

```
                }
        fprintf(stderr, "thread %i has the resource \n", identity);
        finish_with_resource();
        fprintf(stderr, "thread %i is done with the resource \n", identity);
        }
        pthread_barrier_destroy(&barrier);
        calctime[identity]=((double)en)/CLOCKS_PER_SEC;
        printf("Calculation time for thread %d is %f\n",identity,calctime[identity]);
        fprintf(stderr, "thread %i finished\n\n\n", identity);

        return NULL;
}
//END threads----------------------------------------------------------------
int main()
{
        pthread_t queueing;
        init_main_queue();
        int I;
        for (I = 0; I < TOTAL_THREADS; I++)
        {
        //create threads with a delay so they queue in order to start with
        start[I]=clock();
        pthread_create(workers + I, NULL, &worker_thread, &I);
        CPU_ZERO(&cpuset);
        CPU_SET((rand()%3), &cpuset);
        s = pthread_setaffinity_np(workers[I], sizeof(cpu_set_t), &cpuset);
```

```
        sleep(0.01);
        }
        pthread_create(&queueing, NULL, &queuing_thread, NULL);
        for (I = 0; I < TOTAL_THREADS; I++){
        pthread_join(workers[I], NULL);
        }
        pthread_cancel(queueing);
        pthread_join(queueing, NULL);

        fini_main_queue();
        for(int i=0;i<TOTAL_THREADS;i++){
        end[i]=clock();
        printf("Thread %d\n",i+1);
        printf("Creation time: %f\n",((double)start[i])/CLOCKS_PER_SEC);
        printf("End time: %f\n",((double)end[i])/CLOCKS_PER_SEC);
        printf("Latency %f\n\n\n",lat[i]);
  }
}
```

5.2 Validation Source Code for A.M.P

```
#define _GNU_SOURCE
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
```

```
#define QUEUE_SIZE    2000
#define TOTAL_THREADS 500
#define ITERATIONS    1

clock_t start[TOTAL_THREADS];
clock_t end[TOTAL_THREADS];
double calctime[TOTAL_THREADS];
double lat[TOTAL_THREADS];

int s, j;
cpu_set_t cpuset;
pthread_t workers[TOTAL_THREADS];
typedef pthread_barrier_t *waiting_thread;

//struct to simulate fifo-based mutex-----------------------------------------
typedef struct
{
        int             init;
        waiting_thread    threads[QUEUE_SIZE];
        pthread_mutex_t   mutex, change_mutex;
        pthread_cond_t    queue_change;
        pthread_barrier_t queuing_sync;
        unsigned int      front, count, max;
} thread_queue;
static thread_queue main_queue = { 0 };
```

```
//END struct to simulate fifo-based mutex-------------------------------------
//functions for dealing with queue-------------------------------------------
//initialize the queue
static int init_main_queue()
{
        if (main_queue.init) return 1;
        memset(main_queue.threads, 0, sizeof(waiting_thread) * QUEUE_SIZE);
        if (pthread_mutex_init(&main_queue.mutex, NULL) != 0)           return 0;
        if (pthread_mutex_init(&main_queue.change_mutex, NULL) != 0)    return
0;
        if (pthread_cond_init(&main_queue.queue_change, NULL) != 0)     return 0;
        if (pthread_barrier_init(&main_queue.queuing_sync, NULL, 2) != 0) return 0;
        main_queue.front = 0;
        main_queue.count = 0;
        main_queue.max   = QUEUE_SIZE;
        main_queue.init  = 1;
        return 1;
}

//clean up the queue
static int fini_main_queue()
{
        //(there should probably be some sort of 'main_queue.threads' cleanup)

        if (!main_queue.init) return 1;
        pthread_mutex_destroy(&main_queue.mutex);
```

```
	pthread_mutex_destroy(&main_queue.change_mutex);
	pthread_cond_destroy(&main_queue.queue_change);
	pthread_barrier_destroy(&main_queue.queuing_sync);
	return 1;
}
//push a thread onto the queue (from the thread itself)
static int push_waiting_thread(waiting_thread tThread)
{
	//obtain a queue lock
	if (!main_queue.init || pthread_mutex_lock(&main_queue.mutex) != 0) return
0;
	if (main_queue.count == main_queue.max)
	{
	pthread_mutex_unlock(&main_queue.mutex);
	return 0;
	}
	//notify the queuing thread of a change; the lock on the mutex will
	//prevent it from continuing until the change is made
	pthread_cond_broadcast(&main_queue.queue_change);

	//add the thread to the queue
	main_queue.threads[(main_queue.front + main_queue.count++) %
main_queue.max] = tThread;
	pthread_mutex_unlock(&main_queue.mutex);

	return 1;
}
//wait for access to the "resource" (from the thread needing it)
```

```
static int wait_for_resource(waiting_thread tThread, int iIdentity)
{
	//add the thread to the queue and wait for its number to come up
	if (!push_waiting_thread(tThread)) return 0;
	fprintf(stderr, "thread %i waiting\n", iIdentity);
	int outcome = pthread_barrier_wait(tThread);
	return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}
//"unlock" the simulated mutex when finished with it
static int finish_with_resource()
{
	if (!main_queue.init) return 0;
	int outcome = pthread_barrier_wait(&main_queue.queuing_sync);
	return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}
//call the next thread in the queue (from the queuing thread)
static int pop_waiting_thread()
{
	int outcome;
	//obtain a queue lock
	if (!main_queue.init || pthread_mutex_lock(&main_queue.mutex) != 0) return
0;
	//if the queue is empty, block until a thread is added
	if (!main_queue.count)
	{
	pthread_mutex_unlock(&main_queue.mutex);
	//(there is a chance 'pthread_cond_broadcast' will happen between the
```

```
        //line above and the line below; therefore, a 'nanosleep' might be in
        //order here. a 'pthread_barrier_wait' wouldn't be any easier.)
        pthread_mutex_lock(&main_queue.change_mutex);
        pthread_cond_wait(&main_queue.queue_change,
&main_queue.change_mutex);
        pthread_mutex_unlock(&main_queue.change_mutex);
        return 0;
        }
        waiting_thread next = main_queue.threads[main_queue.front++];
        --main_queue.count;
        main_queue.front %= main_queue.max;
        pthread_mutex_unlock(&main_queue.mutex);
        //continue the thread
        //(this will cause a segfault if 'next' has been destroyed already)
        outcome = pthread_barrier_wait(next);
        if (outcome && outcome != PTHREAD_BARRIER_SERIAL_THREAD)
return 0;
        //wait for the thread to finish with the resource
        outcome = pthread_barrier_wait(&main_queue.queuing_sync);
        return !outcome || outcome == PTHREAD_BARRIER_SERIAL_THREAD;
}

//END functions for dealing with queue----------------------------------------
//threads---------------------------------------------------------------------
//queuing thread
static void *queuing_thread(void *iIgnore)
{
        struct timespec delay = { 0, 100 * 1000 * 1000 };
```

```
        if (!main_queue.init) return NULL;
        fprintf(stderr, "queuing thread started\n");
        while (1)
        {
        pop_waiting_thread();
        nanosleep(&delay, NULL);
        }
        fprintf(stderr, "queuing thread finished\n");
        return NULL;
}
void perform_add(){
int a=1000,b=2000;
int sum=a+b;
}
void perform_mul(){
int a=10,b=20;
int prod=a*b;
}
void perform_pi_calc(){
int i;
double p = 0;
for ( i = 1; i <= 10000; i++) {
 if ( i % 2 == 0) {
 p = p - (4.0 / (2 * i - 1));
}
else {
 p = p + (4.0 / (2 * i - 1));
```

```
}
//printf( "%f\t", p );
}
}
//worker threads
static void *worker_thread(void *iIdentity)
{
        int identity = *(int*) iIdentity, iterations = ITERATIONS;
        clock_t temp,st,en;
        if(lat[identity]==0){
        temp=clock()-start[identity];
        lat[identity]=((double)temp)/CLOCKS_PER_SEC;
        }
        fprintf(stderr, "thread %i started\n", identity);
        pthread_barrier_t barrier;
        if (pthread_barrier_init(&barrier, NULL, 2) != 0) return NULL;
        while (iterations-- > 0 && wait_for_resource(&barrier, identity))
        {
        s = pthread_getaffinity_np(workers[identity], sizeof(cpu_set_t), &cpuset);
        for (j = 0; j < CPU_SETSIZE; j++)
          if (CPU_ISSET(j, &cpuset)){
              printf("THREAD %d USING CPU %d\n",identity,j);
                        if(j==0){
                                st=clock();
                                perform_add();
                                en=clock()-st;
                        }
```

```
                else if(j==1){
                        st=clock();
                        perform_mul();
                        en=clock()-st;
                }
                else{
                        st=clock();
                        perform_pi_calc();
                        en=clock()-st;
                }
        }
        fprintf(stderr, "thread %i has the resource \n", identity);
        finish_with_resource();
        fprintf(stderr, "thread %i is done with the resource \n", identity);
        }
        pthread_barrier_destroy(&barrier);
        calctime[identity]=((double)en)/CLOCKS_PER_SEC;
        printf("Calculation time for thread %d is %f\n",identity,calctime[identity]);
        fprintf(stderr, "thread %i finished\n\n\n", identity);
        return NULL;
}
//END threads----------------------------------------------------------------
int main()
{
        pthread_t queueing;
        init_main_queue();
        int I;
```

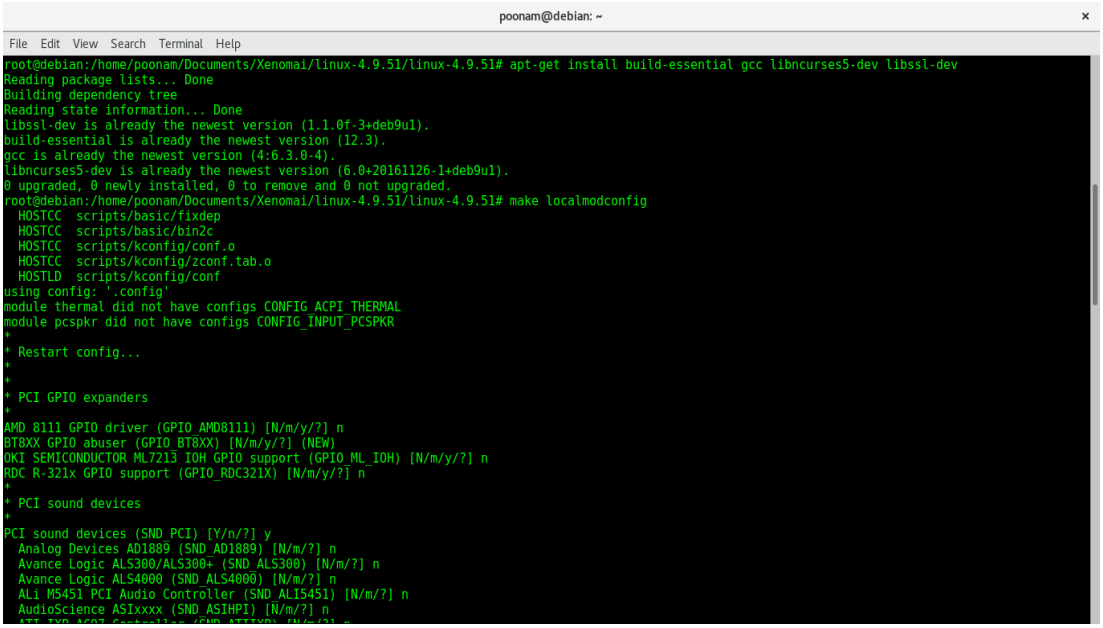
```
        for (I = 0; I < TOTAL_THREADS; I++)
        {
        //create threads with a delay so they queue in order to start with
        start[I]=clock();
        pthread_create(workers + I, NULL, &worker_thread, &I);
        CPU_ZERO(&cpuset);
        CPU_SET((rand()%3), &cpuset);
        s = pthread_setaffinity_np(workers[I], sizeof(cpu_set_t), &cpuset);
        sleep(0.01);
        }
        pthread_create(&queueing, NULL, &queuing_thread, NULL);
        for (I = 0; I < TOTAL_THREADS; I++){
        pthread_join(workers[I], NULL);
        }
        pthread_cancel(queueing);
        pthread_join(queueing, NULL);
        fini_main_queue();
        for(int i=0;i<TOTAL_THREADS;i++){

        end[i]=clock();
        printf("Thread %d\n",i+1);
        printf("Creation time: %f\n",((double)start[i])/CLOCKS_PER_SEC);
        printf("End time: %f\n",((double)end[i])/CLOCKS_PER_SEC);
        printf("Latency %f\n\n\n",lat[i]);
    }
}
```

CHAPTER-6

RESULTS

Building essential packages in Linux

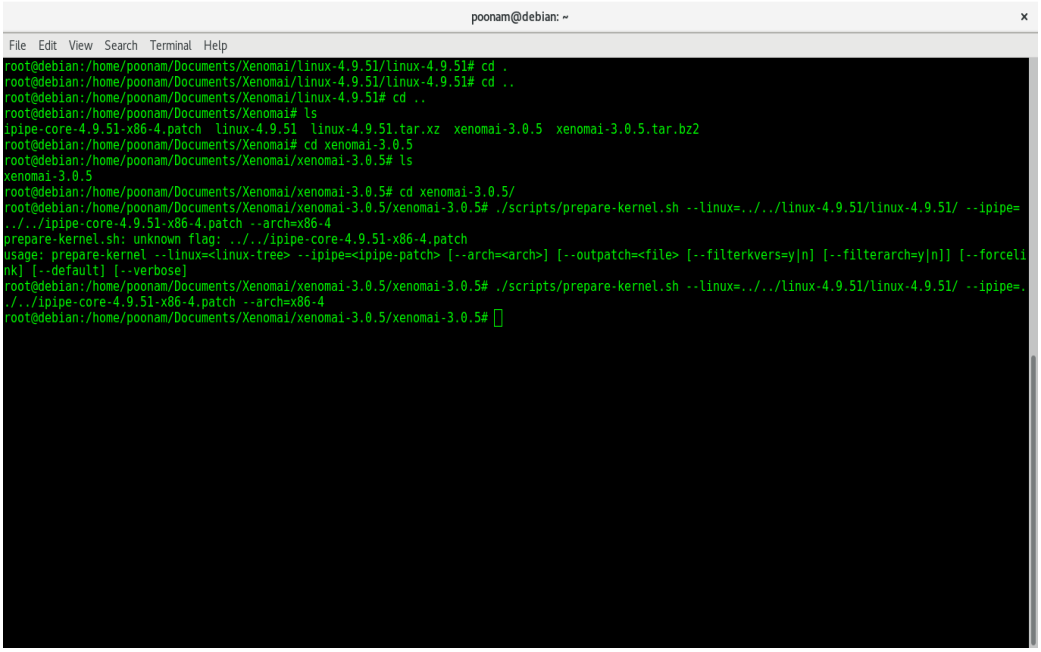
First we need to build the essential packages, The libncurses5-dev package will make it simpler to design the kernel, so make a point to introduce it.



Kernel Patching:

Now we need to patch our downloaded stable Linux kernel with Xenomai using

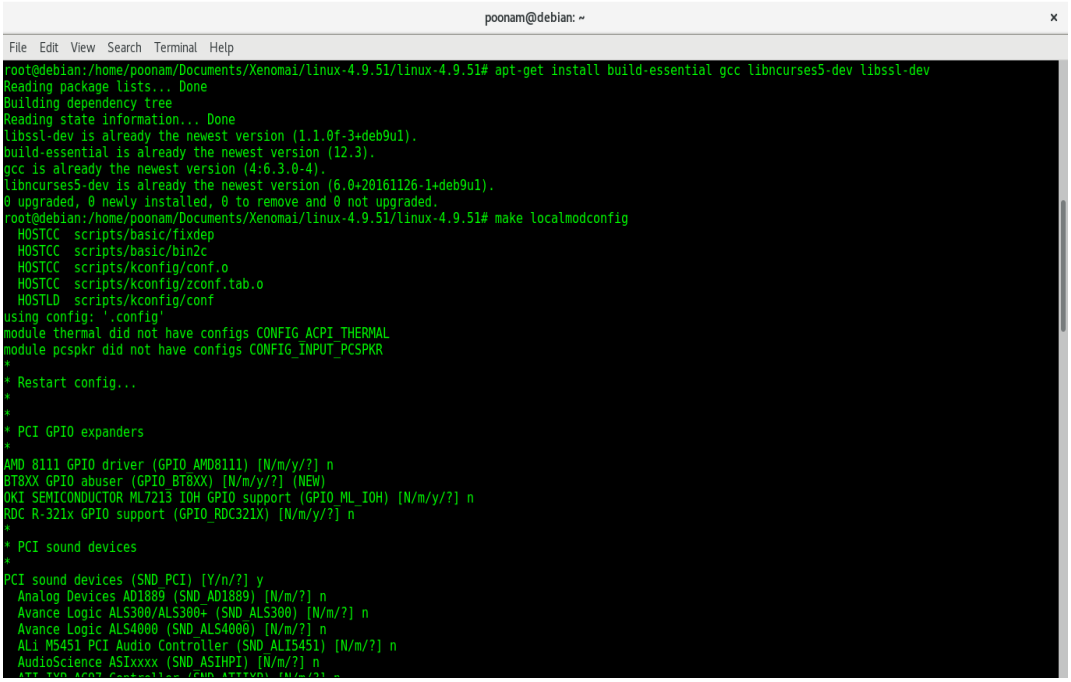
"/scripts/prepare-kernel.sh-Linux=../../Linux-4.9.51/Linux-4.9.51/--ipipe=../../ipipe-core- 4.9.51-x86-4.patch-arch=x86-4"



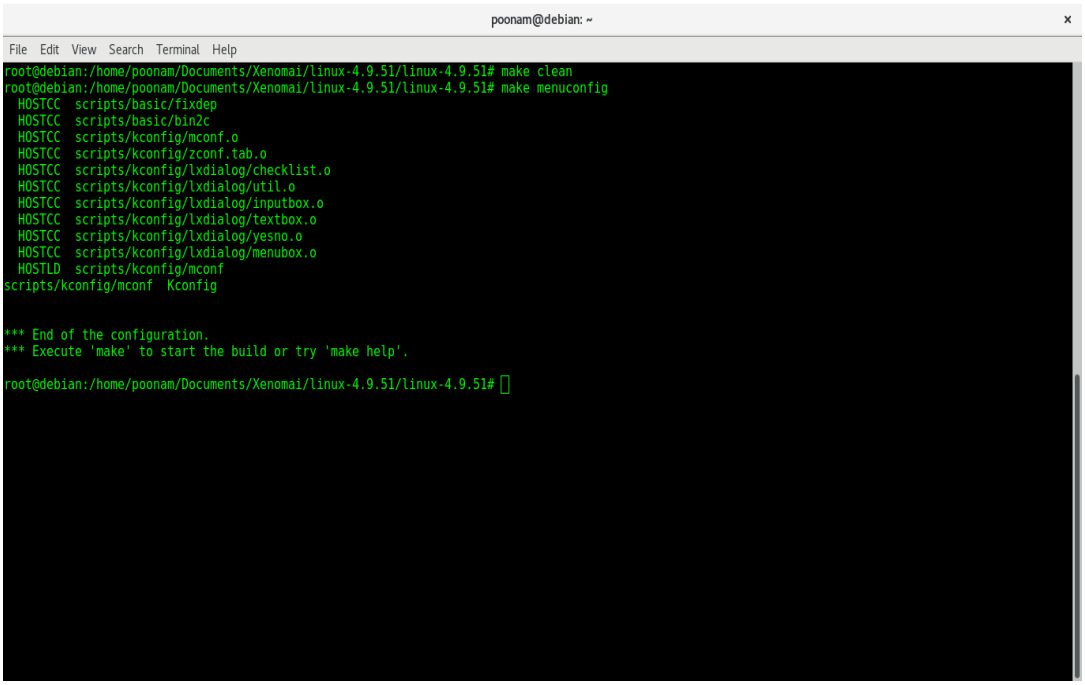
Configuring Patched kernel:

"make localmodconfig"

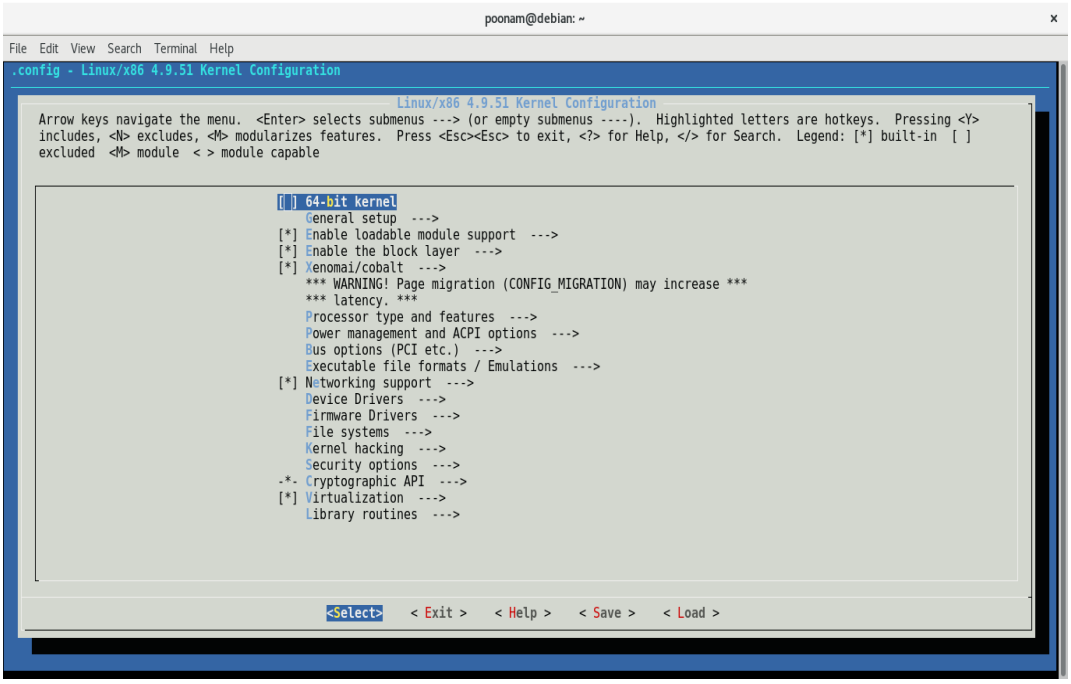
Using **"make localmodconfig"** detects currently running kernel components and marks them for compilation.



"make menuconfig" is used to choose what to compile. Here we make changes and generate a .config file.



.config -Linux/x86 4.9.51 kernel configuration file



Configuration of kernel :

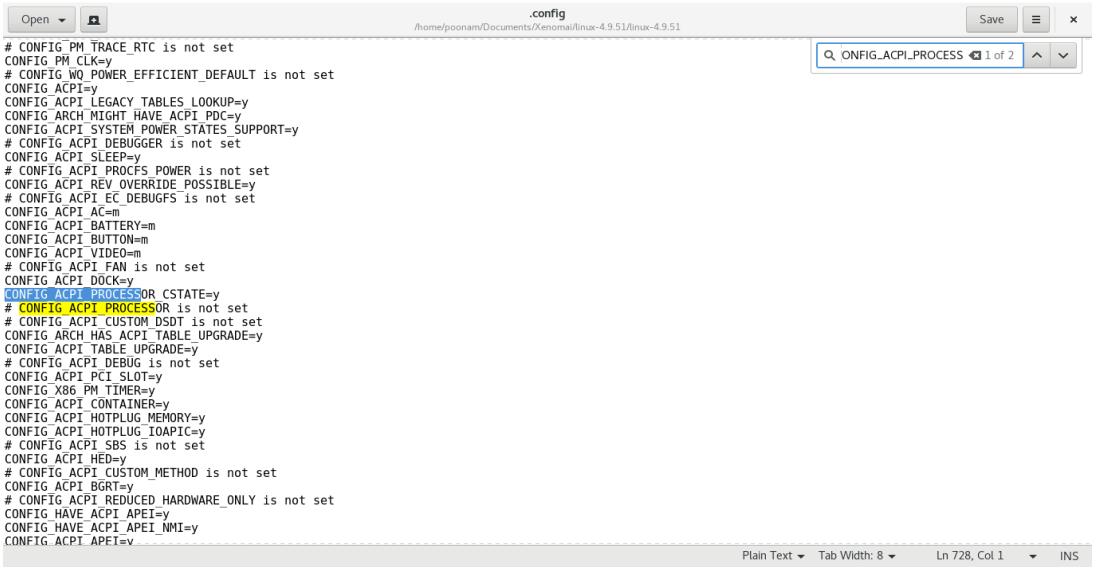
During configuring we need to enable and disable some settings

CONFIG_APM- disable

Power management control to the BIOS is assigned by the APM model. BIOS code is never composed in light of best inertness. APM schedules are conjured with SMI need if designed, this sidesteps the I-pipe totally.

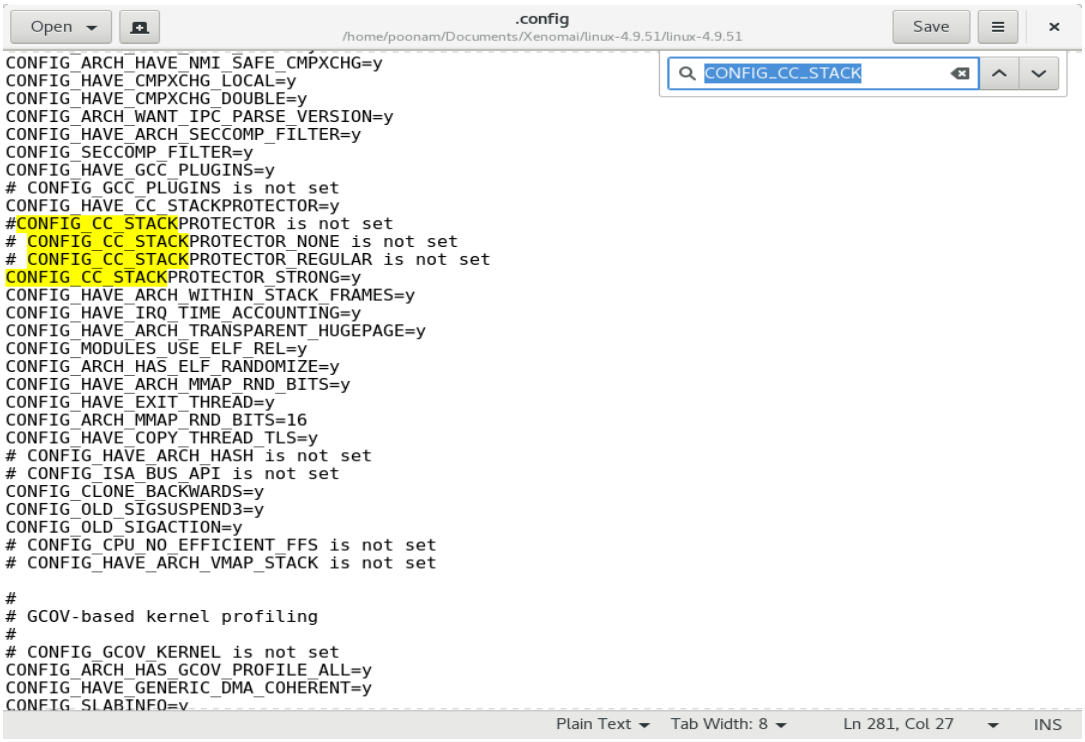
CONFIG_ACPI_PROCESSOR-disable

The ACPI processor module disables the local ACPI which will cause the Xenomai timer initialization to fail due to this reason we disable this option.



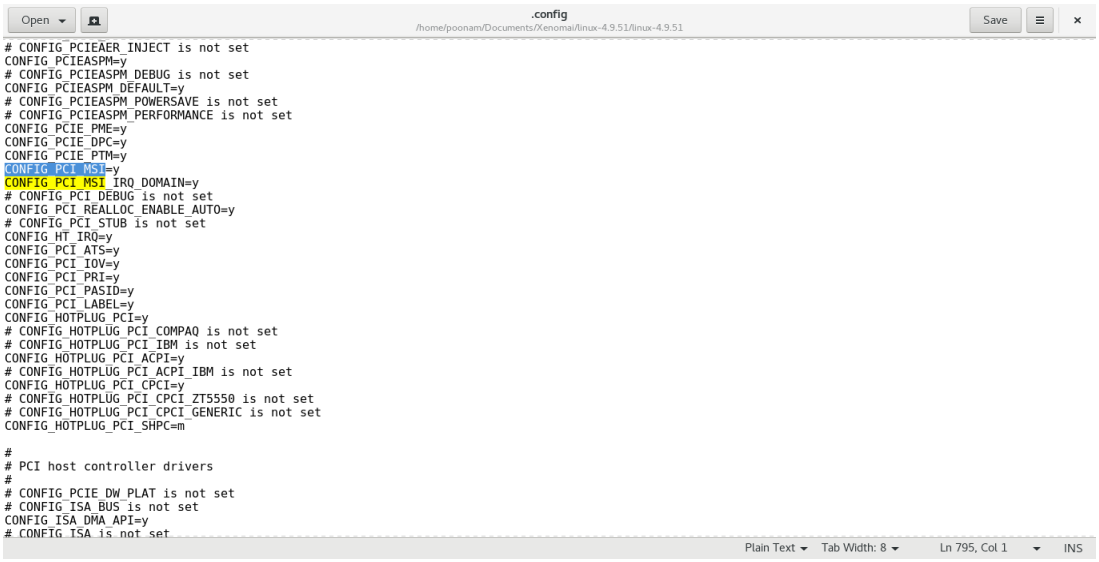
CONFIG_CC_STACKPROTECTOR-disable

This option may be enabled on x86_64 only with Xenomai 2, x86_64 and x86_32 indifferently with Xenomai 3.



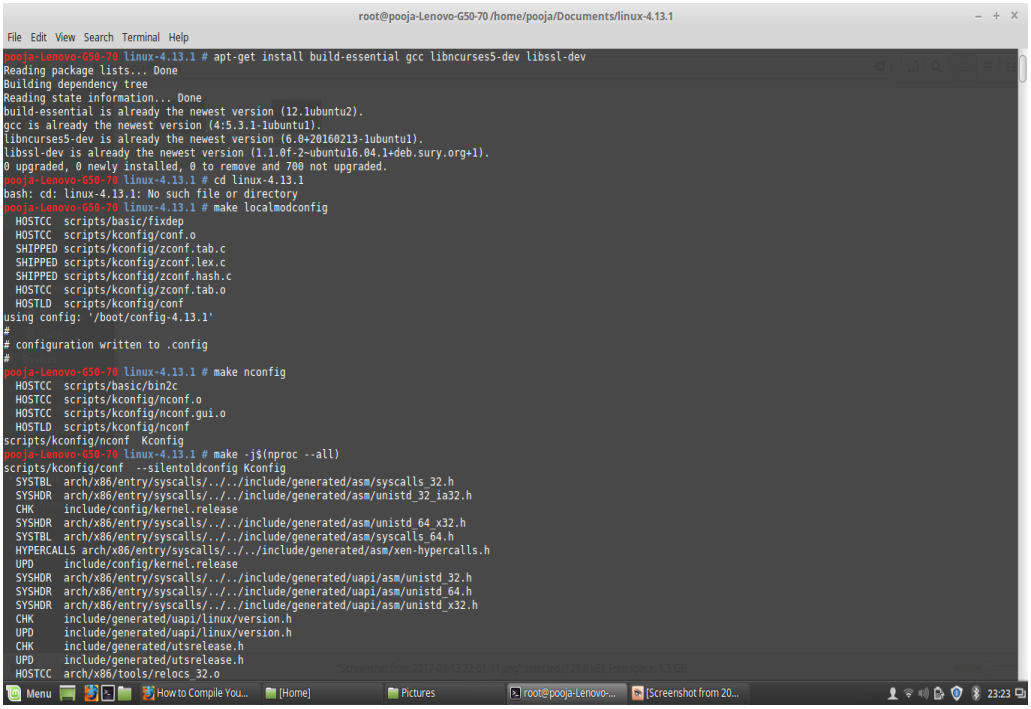
CONFIG_PCI_MSI:

Enabling this config_pci_msi will always make operations like hooking, releasing, disabling and enabling to be always done on Linux kernel and never from the real-time mode.



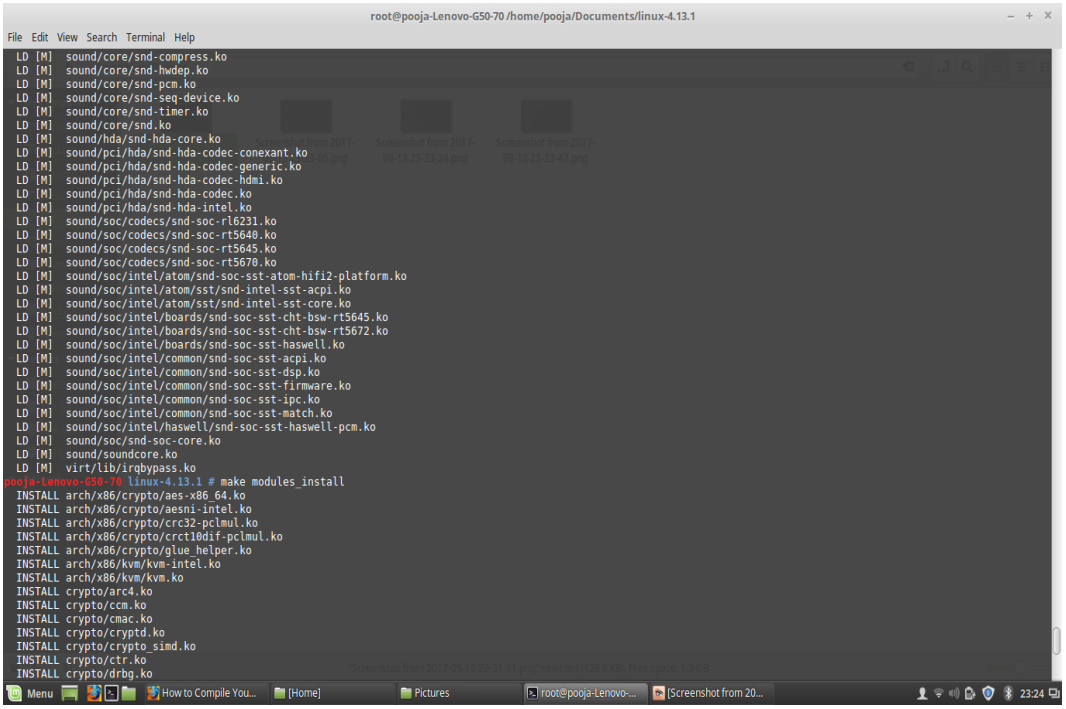
“sudo make -j$(nproc --all)”

This command is used to compile the kernel."$" represents number of processors**.**



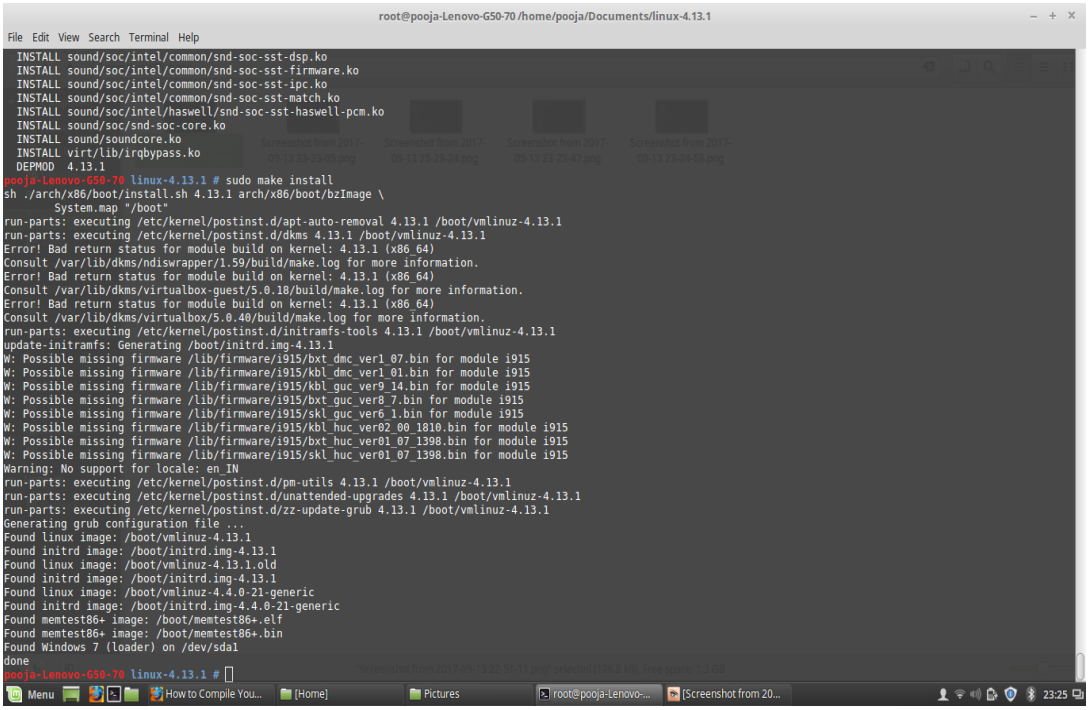
“make modules_install”

This command is used to finish compiling the rest of the kernel.



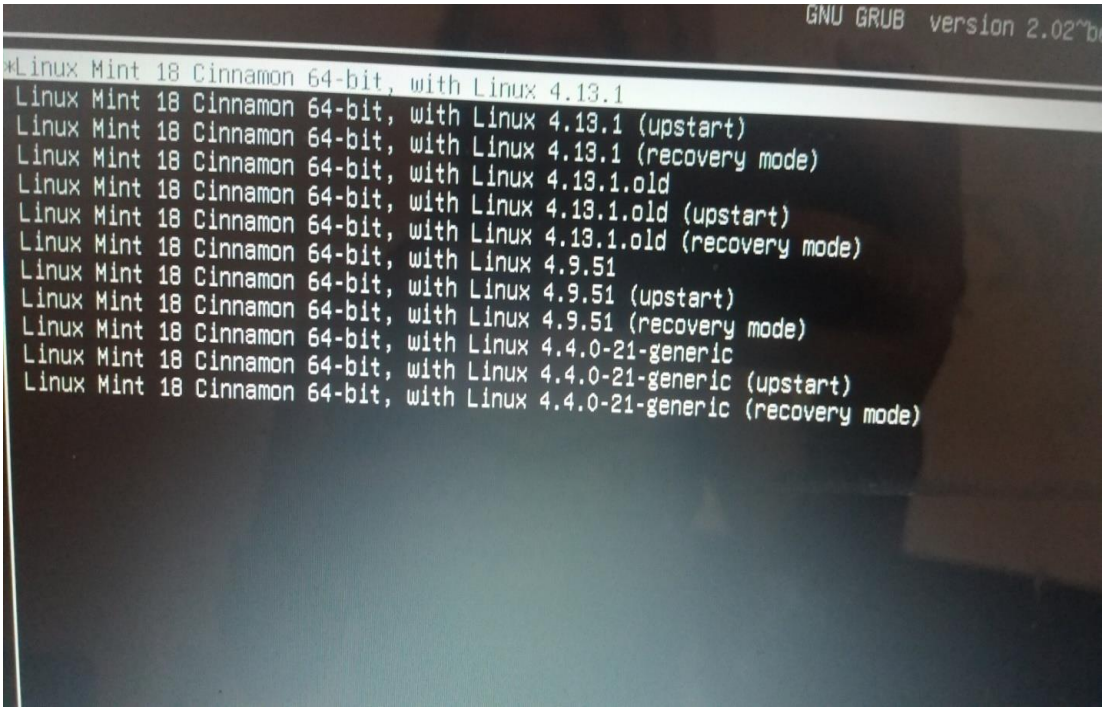
“make install”

This will automatically copy the kernel to your /boot folder and generate the appropriate files to make it work. It updates the grub menu with our compiled kernel.



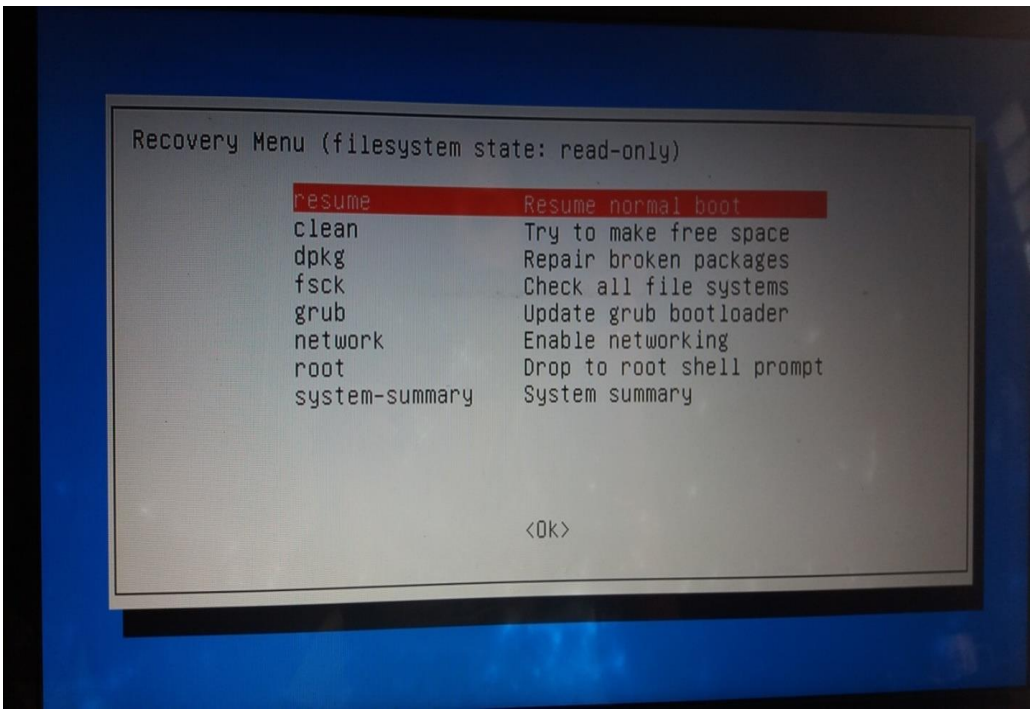
Available kernels after compilation:

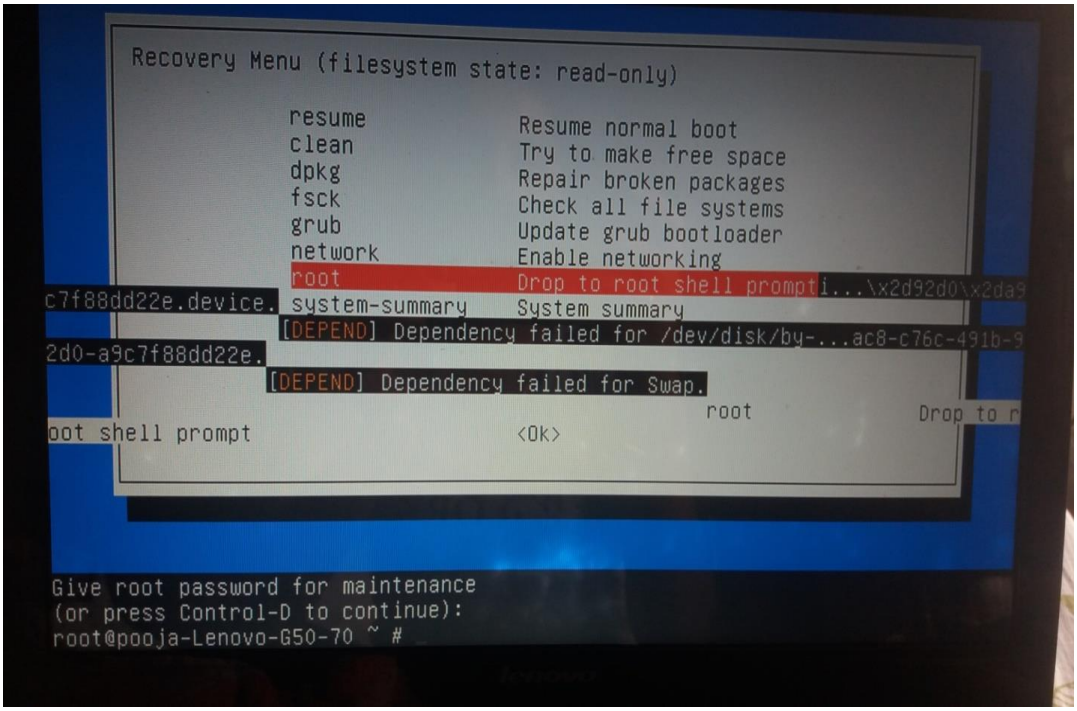
These are the available kernel which will be displayed in our booting options after compilation.



Linux recovery mode

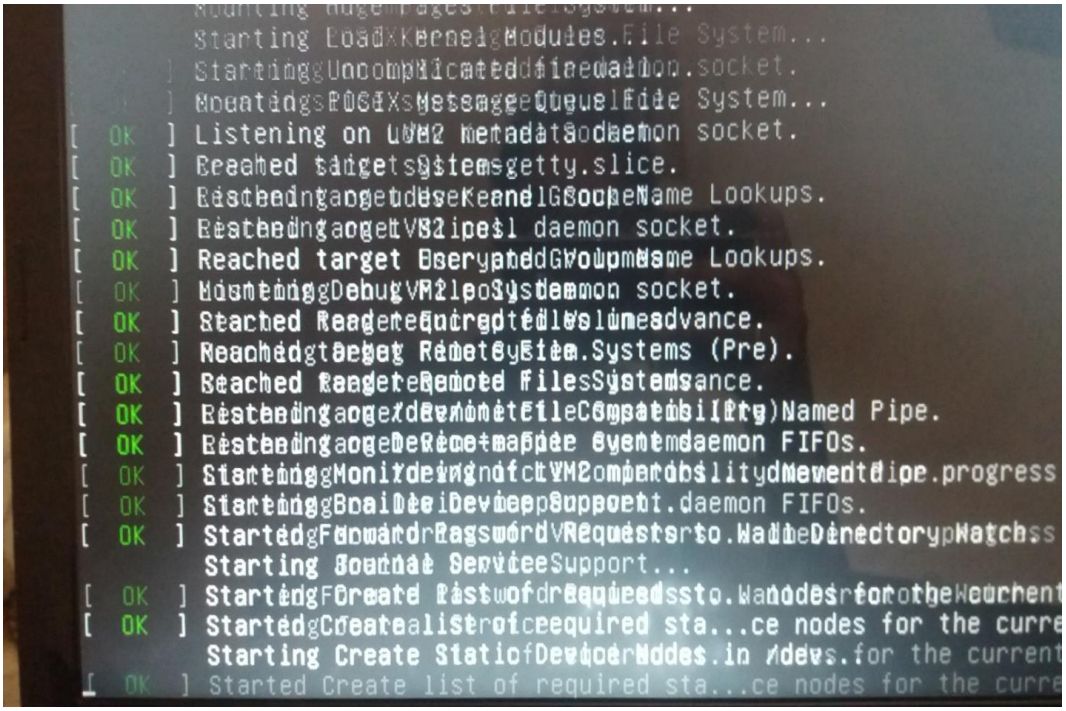
This is the recovery mode of our built Linux kernel which is having file system state as read-only.





Compiled Linux kernel:

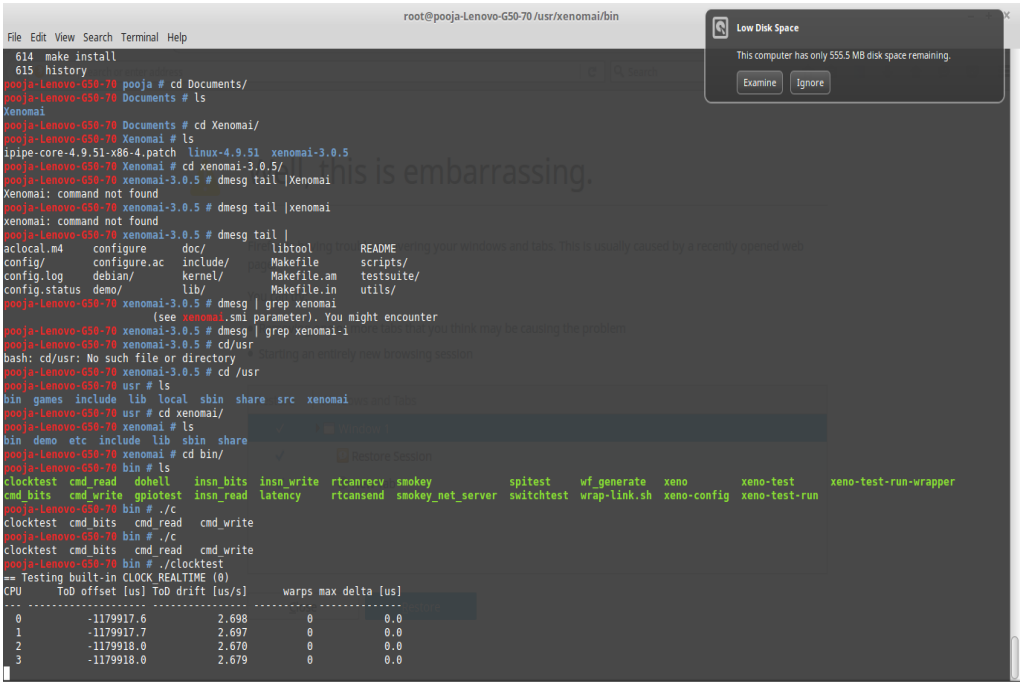
This is the Compiled Linux kernel which we got after all the configurations and compilations.



Functional testing

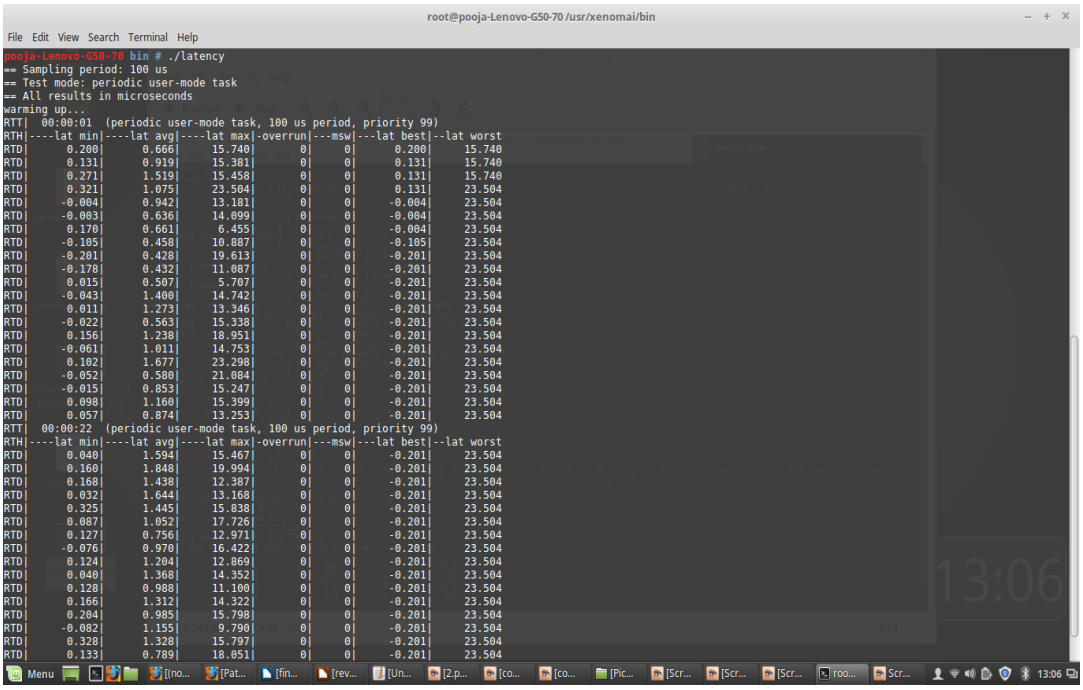
Search for Xenomai:

Before performing the functional tests, the first step is to search for the Xenomai whether it is present in our kernel or not?



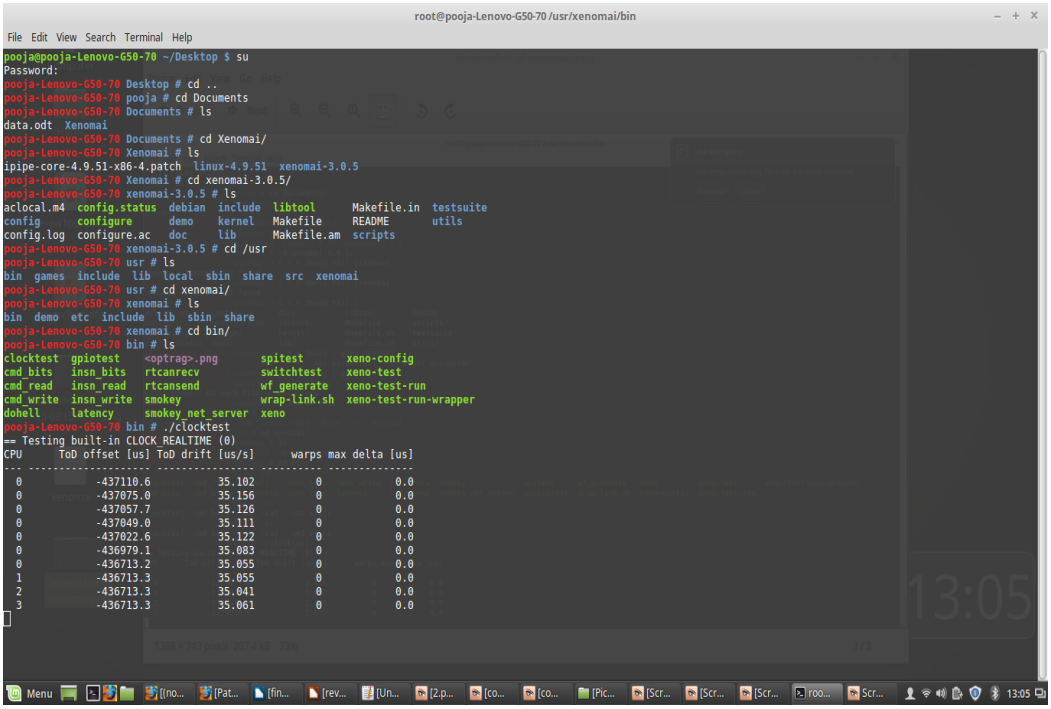
Latency:

Performing Latency test using ./latency



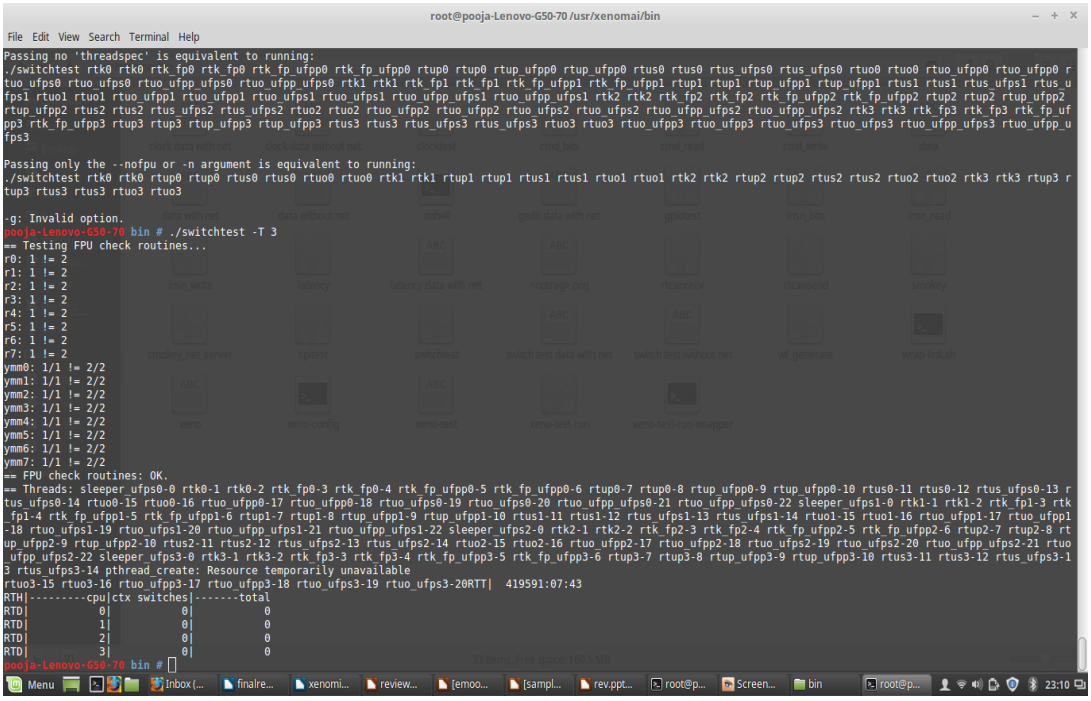
Clock test:

Performing Clock test using ./clocktest



Switch Test:

Performing switch test using ./switchtest.



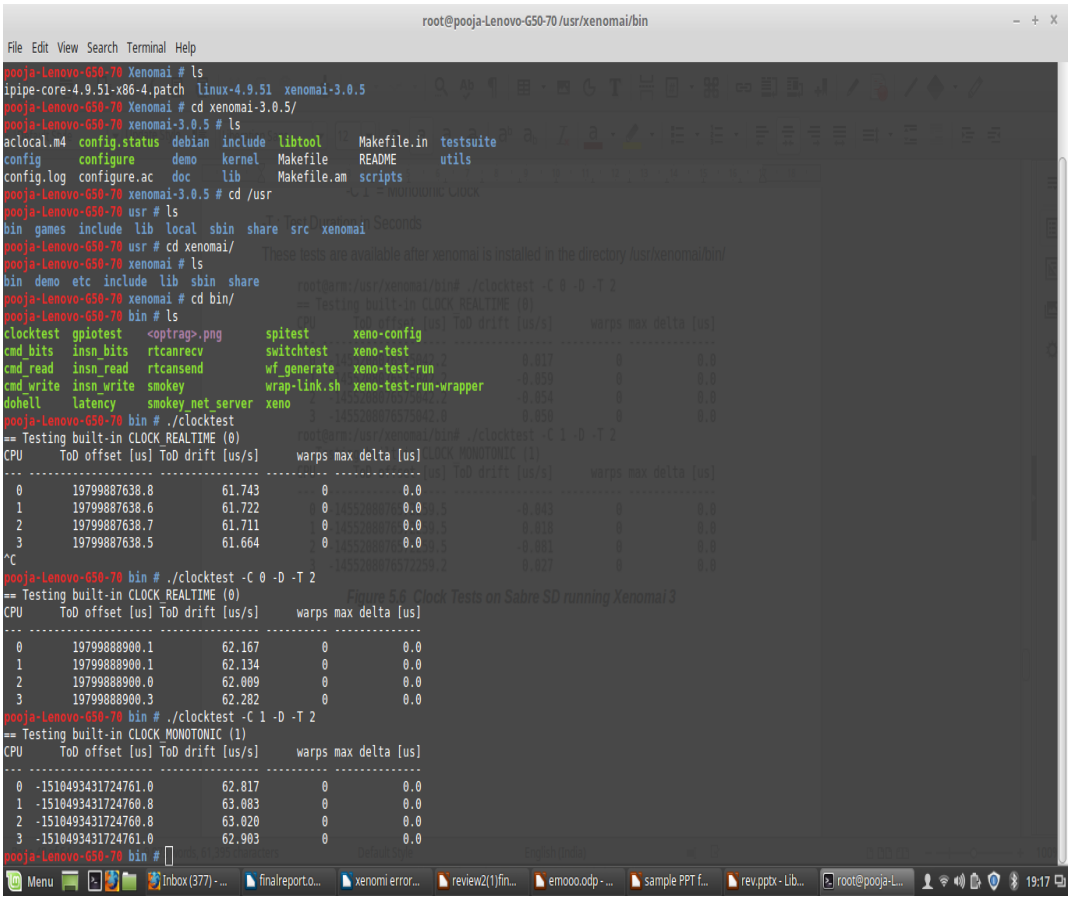
Arguments for Clock test

-C : Clock ID

Usage: -C0 = Real time Clock

-C1 = Monotonic Clock

-T : Test Duration in Seconds



Arguments for Latency Test

-s : Prints Statistics of min, max and average latencies

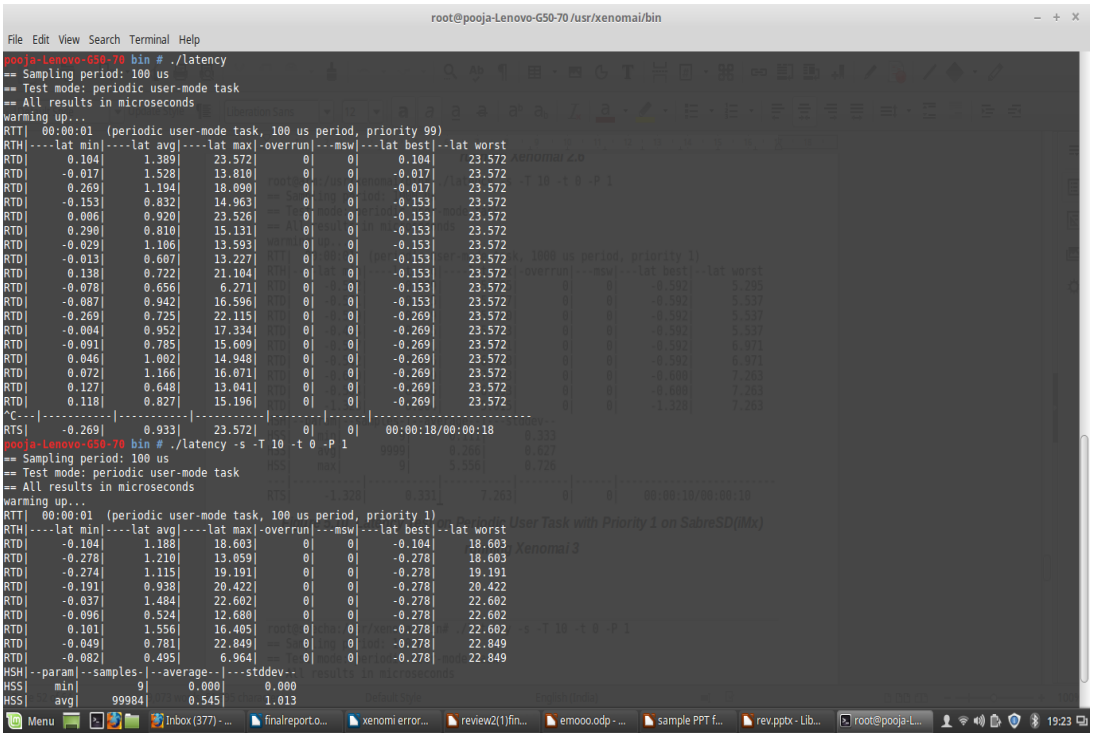
-T : Test duration in seconds

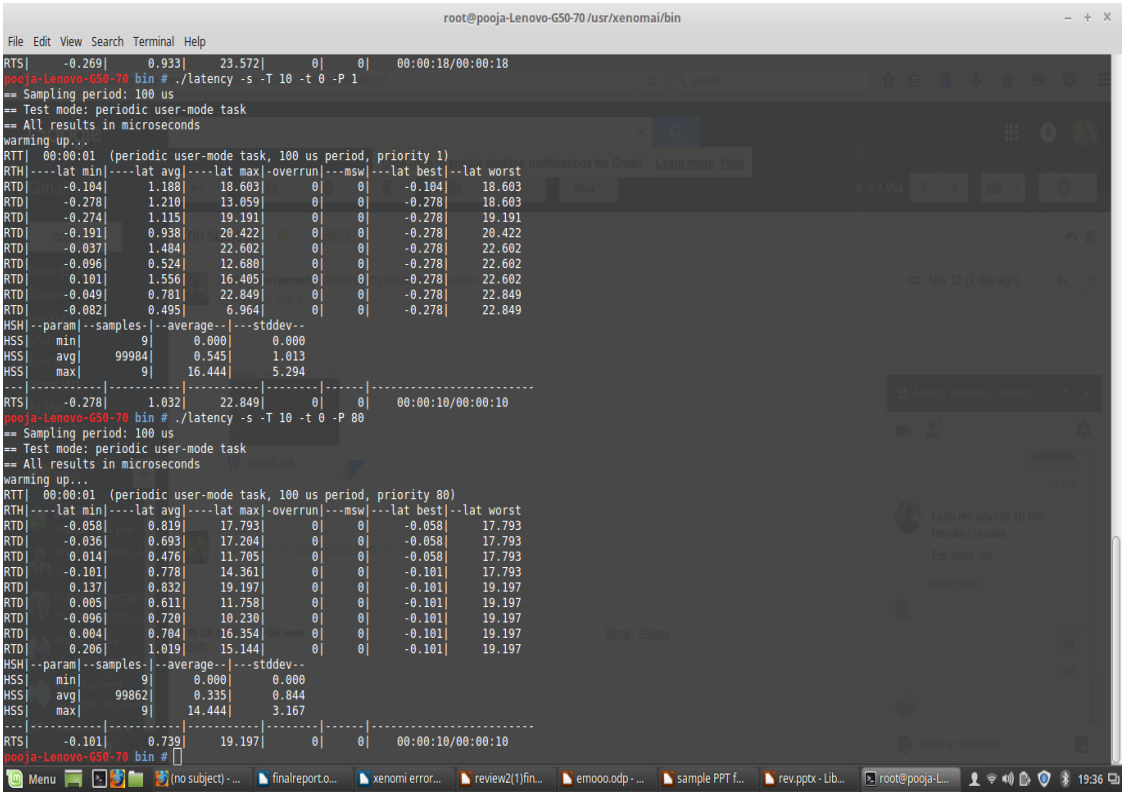
-t : 0 = User task

1= Kernel Task

2 = Timer Task

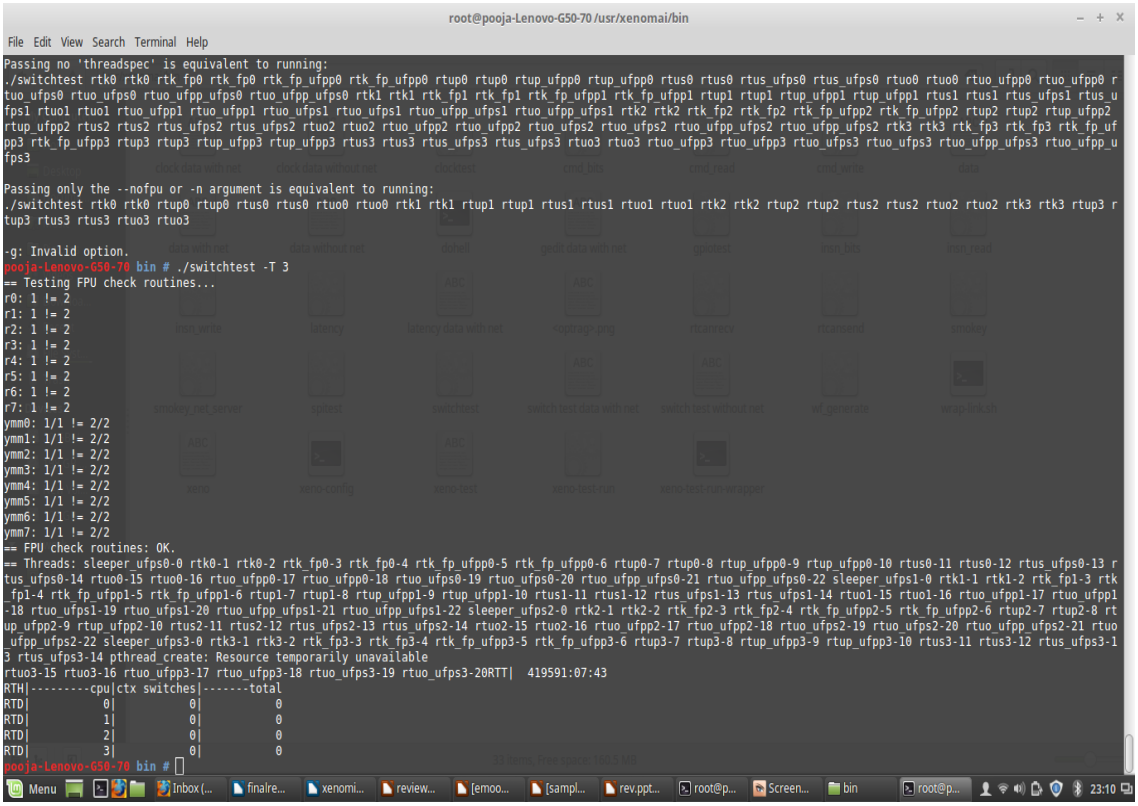
-P : Priority of the Task


root@pooja-Lenovo-G50-70 /usr/xenomai/bin
File Edit View Search Terminal Help
pooja-Lenovo-G50-70 bin # ./latency
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| 0.104| 1.389| 23.572| 0| 0| 0.104| 23.572
RTD| -0.017| 1.528| 13.810| 0| 0| -0.017| 23.572
RTD| 0.269| 1.194| 18.090| 0| 0| -0.017| 23.572
RTD| -0.153| 0.832| 14.963| 0| 0| -0.153| 23.572
RTD| 0.006| 0.920| 23.526| 0| 0| -0.153| 23.572
RTD| 0.290| 0.810| 15.131| 0| 0| -0.153| 23.572
RTD| -0.029| 1.106| 13.593| 0| 0| -0.153| 23.572
RTD| -0.013| 0.607| 13.227| 0| 0| -0.153| 23.572
RTD| 0.138| 0.722| 21.104| 0| 0| -0.153| 23.572
RTD| -0.078| 0.656| 6.271| 0| 0| -0.153| 23.572
RTD| -0.087| 0.942| 16.596| 0| 0| -0.153| 23.572
RTD| -0.269| 0.725| 22.115| 0| 0| -0.269| 23.572
RTD| -0.084| 0.952| 17.334| 0| 0| -0.269| 23.572
RTD| -0.091| 0.785| 15.609| 0| 0| -0.269| 23.572
RTD| 0.046| 1.002| 14.948| 0| 0| -0.269| 23.572
RTD| 0.072| 1.166| 16.071| 0| 0| -0.269| 23.572
RTD| 0.127| 0.648| 13.041| 0| 0| -0.269| 23.572
RTD| 0.118| 0.827| 15.196| 0| 0| -0.269| 23.572
^C---|----------|----------|----------|-------|------|------------------------
RTS| -0.269| 0.933| 23.572| 0| 0| 00:00:18/00:00:18
pooja-Lenovo-G50-70 bin # ./latency -s -T 10 -t 0 -P 1
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 1)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| -0.104| 1.188| 18.603| 0| 0| -0.104| 18.603
RTD| -0.278| 1.210| 13.059| 0| 0| -0.278| 18.603
RTD| -0.274| 1.115| 19.191| 0| 0| -0.278| 19.191
RTD| -0.191| 0.938| 20.422| 0| 0| -0.278| 20.422
RTD| -0.037| 1.484| 22.602| 0| 0| -0.278| 22.602
RTD| -0.096| 0.524| 12.680| 0| 0| -0.278| 22.602
RTD| 0.101| 1.556| 16.405| 0| 0| -0.278| 22.602
RTD| -0.049| 0.781| 22.849| 0| 0| -0.278| 22.849
RTD| -0.082| 0.495| 6.964| 0| 0| -0.278| 22.849
HSH|--param|--samples-|--average--|---stddev--
HSS| min| 9| 0.000| 0.000
HSS| avg| 99984| 0.545| 1.013
Menu
Inbox (377) - ...
finalreport.o...
xenomi error...
review2(1)fin...
emooo.odp - ...
sample PPT f...
rev.pptx - Lib...
root@pooja-L...
19:23


root@pooja-Lenovo-G50-70 /usr/xenomai/bin
File Edit View Search Terminal Help
RTS| -0.269| 0.933| 23.572| 0| 0| 00:00:18/00:00:18
pooja-Lenovo-G50-70 bin # ./latency -s -T 10 -t 0 -P 1
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 1)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| -0.104| 1.188| 18.603| 0| 0| -0.104| 18.603
RTD| -0.278| 1.210| 13.059| 0| 0| -0.278| 18.603
RTD| -0.274| 1.115| 19.191| 0| 0| -0.278| 19.191
RTD| -0.191| 0.938| 20.422| 0| 0| -0.278| 20.422
RTD| -0.037| 1.484| 22.602| 0| 0| -0.278| 22.602
RTD| -0.096| 0.524| 12.680| 0| 0| -0.278| 22.602
RTD| 0.101| 1.556| 16.405| 0| 0| -0.278| 22.602
RTD| -0.049| 0.781| 22.849| 0| 0| -0.278| 22.849
RTD| -0.082| 0.495| 6.964| 0| 0| -0.278| 22.849
HSH|--param|--samples-|--average--|---stddev--
HSS| min| 9| 0.000| 0.000
HSS| avg| 99984| 0.545| 1.013
HSS| max| 9| 16.444| 5.294
---|----------|----------|----------|-------|------|------------------------
RTS| -0.278| 1.032| 22.849| 0| 0| 00:00:10/00:00:10
pooja-Lenovo-G50-70 bin # ./latency -s -T 10 -t 0 -P 80
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 80)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| -0.058| 0.819| 17.793| 0| 0| -0.058| 17.793
RTD| -0.036| 0.693| 17.204| 0| 0| -0.058| 17.793
RTD| 0.014| 0.476| 11.705| 0| 0| -0.058| 17.793
RTD| -0.101| 0.778| 14.361| 0| 0| -0.101| 17.793
RTD| 0.137| 0.832| 19.197| 0| 0| -0.101| 19.197
RTD| 0.005| 0.611| 11.758| 0| 0| -0.101| 19.197
RTD| -0.096| 0.720| 10.230| 0| 0| -0.101| 19.197
RTD| 0.004| 0.704| 16.354| 0| 0| -0.101| 19.197
RTD| 0.206| 1.019| 15.144| 0| 0| -0.101| 19.197
HSH|--param|--samples-|--average--|---stddev--
HSS| min| 9| 0.000| 0.000
HSS| avg| 99862| 0.335| 0.844
HSS| max| 9| 14.444| 3.167
---|----------|----------|----------|-------|------|------------------------
RTS| -0.101| 0.739| 19.197| 0| 0| 00:00:10/00:00:10
pooja-Lenovo-G50-70 bin #
Menu
(no subject) - ...
finalreport.o...
xenomi error...
review2(1)fin...
emooo.odp - ...
sample PPT f...
rev.pptx - Lib...
root@pooja-L...
19:36

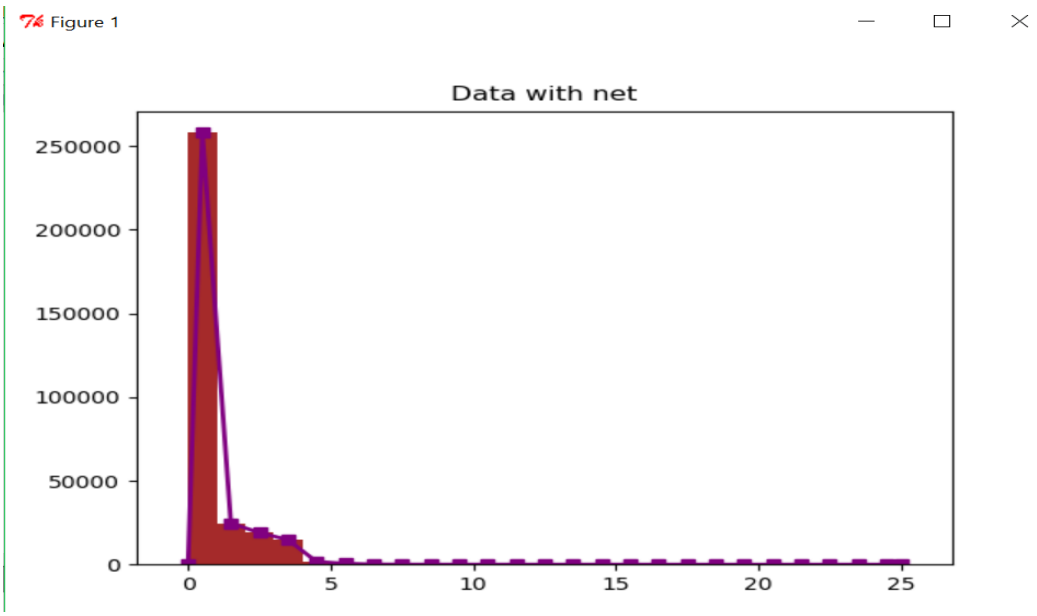
Arguments for Switch test

T- Time duration for every second.



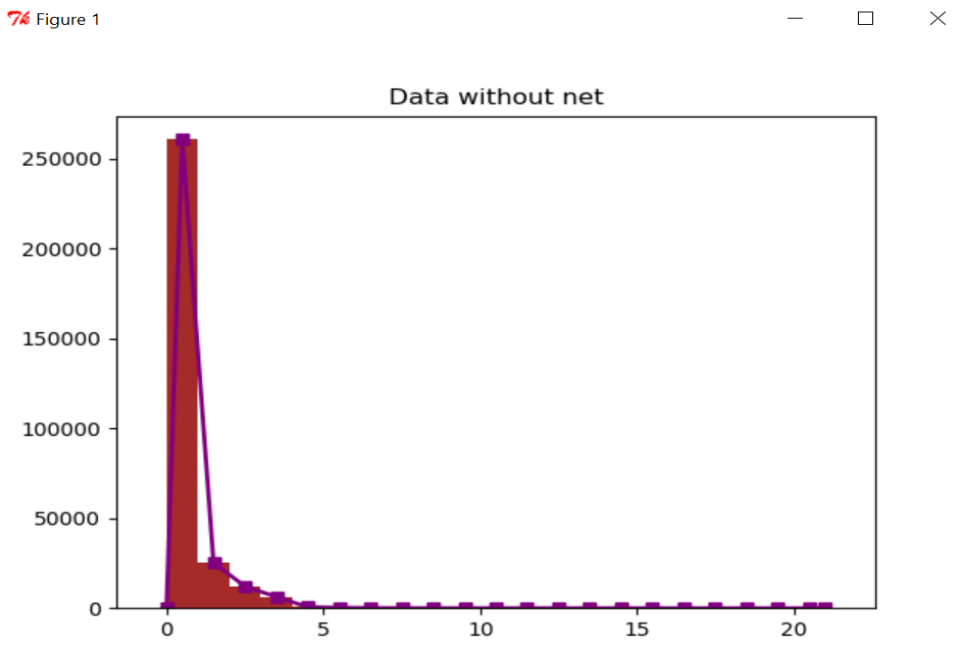
Graph showing latency values when some task given:

Give a task to CPU (example downloading some files) and perform the latency test.



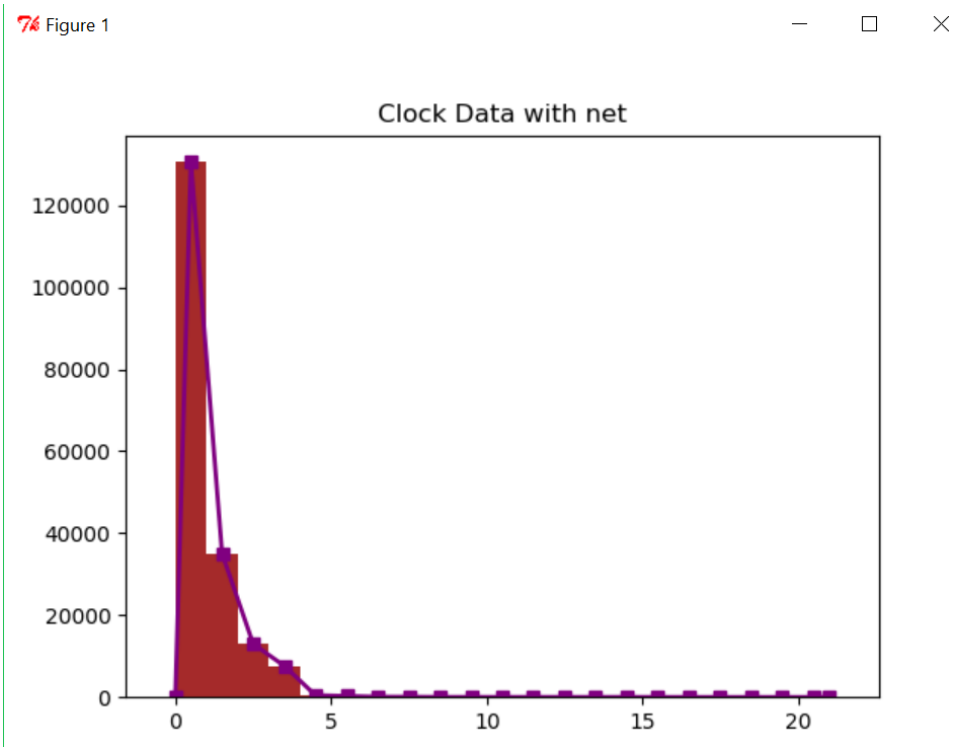
Graph showing latency values without any task given :

Check the same without giving any task to CPU.



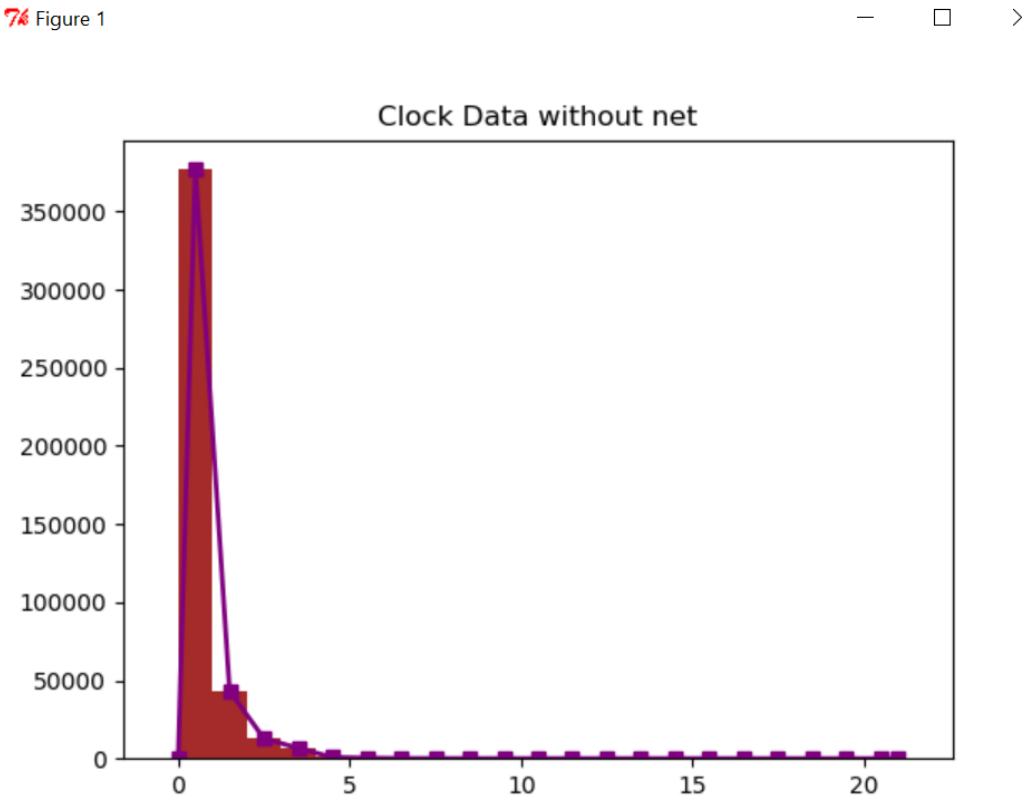
Graph showing clock values with some task given:

Give a task to CPU (example downloading some files) and perform the clock test.



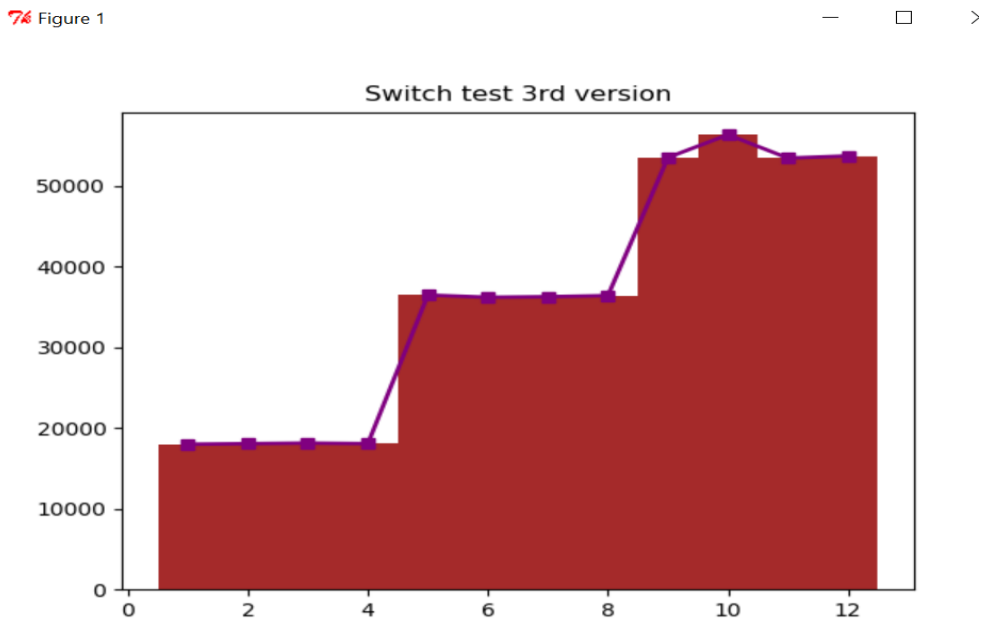
Graph showing clock values without any task given:

Check the same without giving any task to CPU.



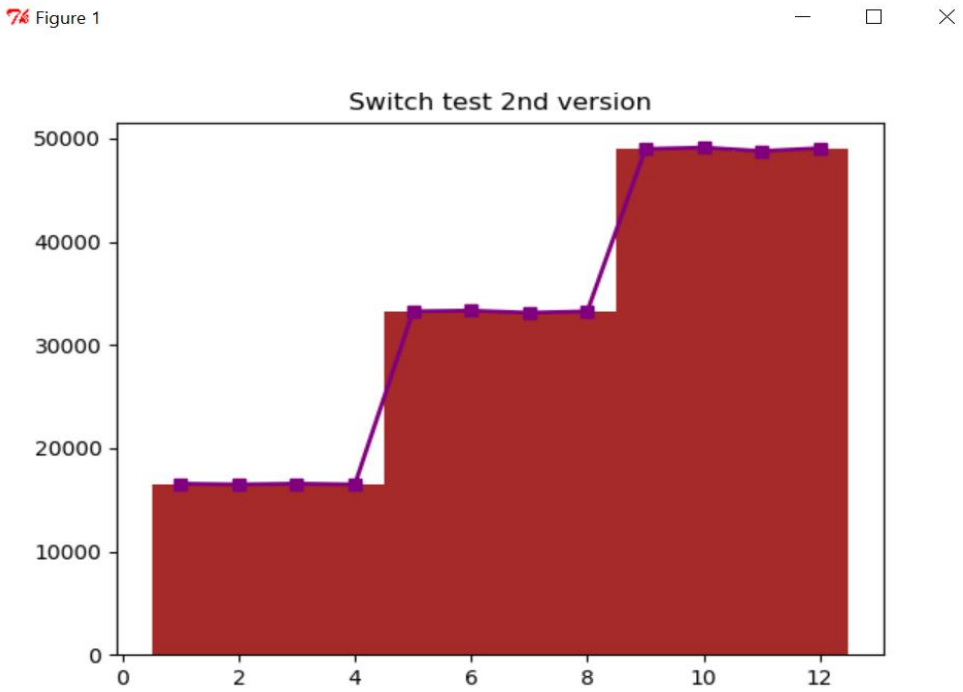
Graph showing switch test values with some task given

Give a task to CPU (example downloading some files) and perform the switch test.



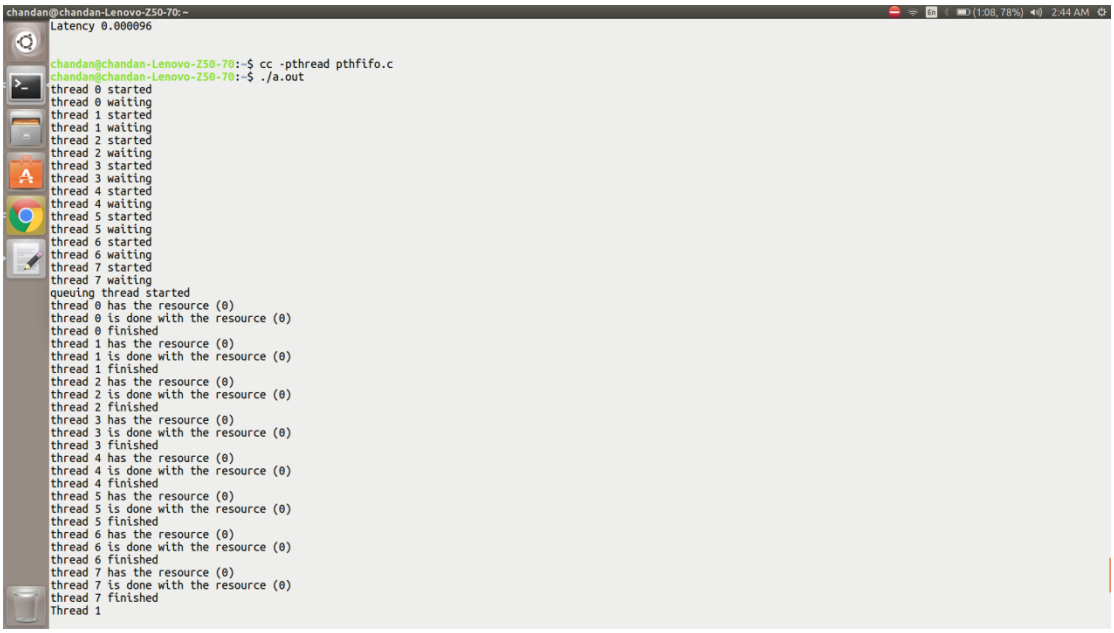
Graph showing switch test values without any task given:

Check the same without giving any task to CPU



Asynchronous Processing on Multi Core Processors

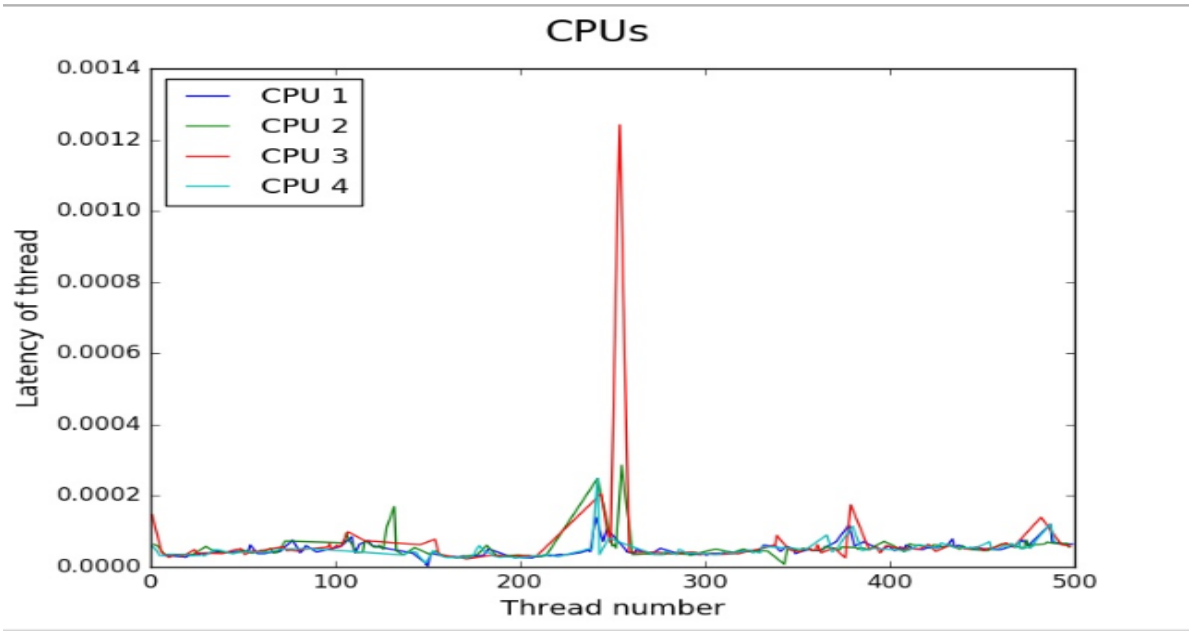
Various threads were created and each thread is given a specific task.



Creation of threads latency time and end time of each thread.

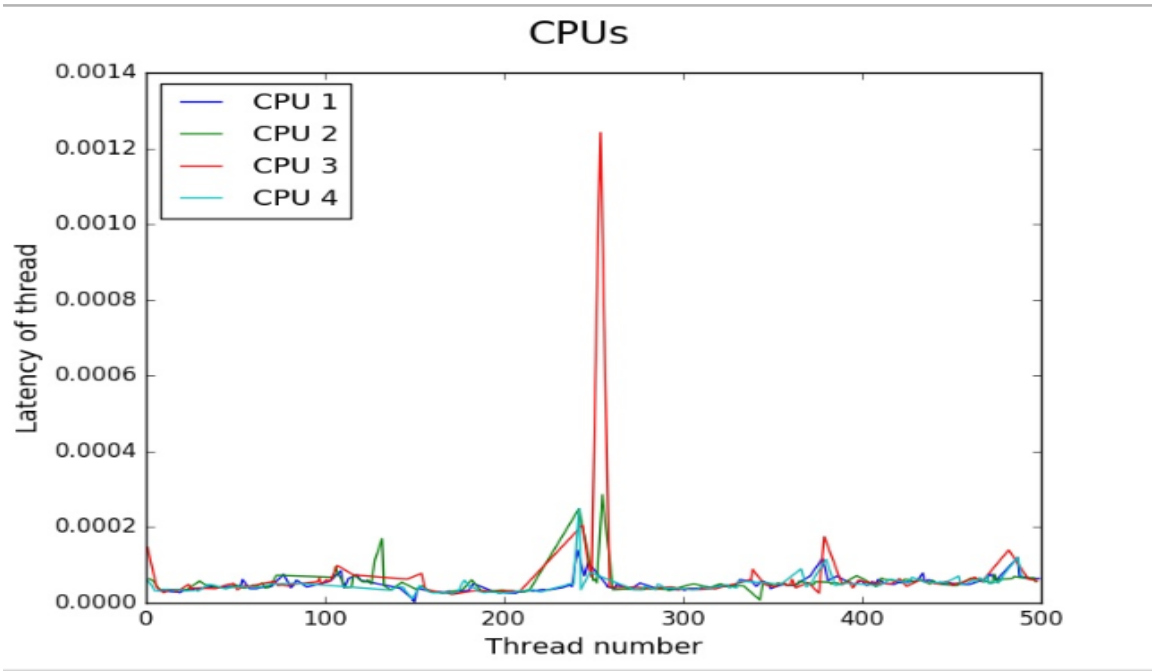
Asynchronous Processing on standard Linux kernel (AMD II X3 720 Processor, Frequency- 800 MHz

Asynchronous Processing plot on standard Linux kernel (AMD II X3 720 Processor, Frequency- 800 MHz



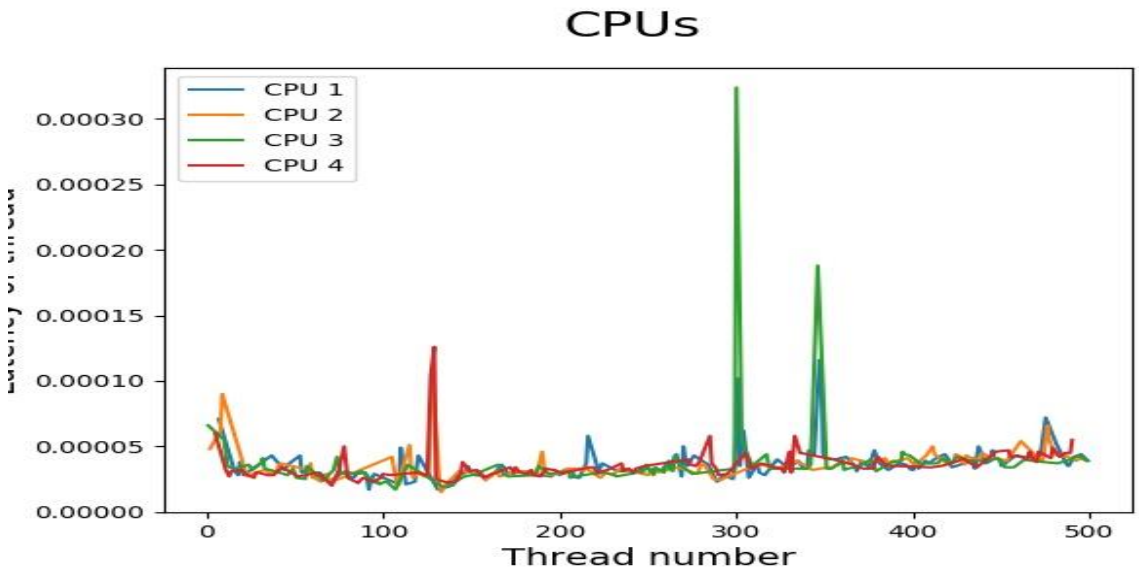
Asynchronous process on standard Linux kernel (x86 i3 processor, Frequency- 1900 MHz)

Asynchronous Processing plot on standard Linux kernel (x86 i3 processor, Frequency- 1900 MHz



Asynchronous Processing plot on standard Linux kernel (x86 i5 processor, Frequency-1400.158 MHz).

Asynchronous Processing plot on standard Linux kernel (x86 i5 processor, Frequency-1400.158 MHz).



CHAPTER-7

DISCUSSION ON RESULT

We took a stable Linux Kernel and downloaded stable Xenomai version, collected a stable I-PIPE core to patch with Xenomai. Essential packages were built in the stable Linux kernel. Later Linux kernel was patched with Xenomai and configuration was done. Then compiled it and functional tests like Latency,Switch and clock tests were performed to measure the performance.

We have implemented asymmetric multi processing on various multi core processors like a. x86 i3 processor with Frequency- 1900 MHz. b. AMD II X3 720 Processor, Frequency- 800 MHz and c. x8 i5 Processor, Frequency- 1400.158 MHz. The system performance and latency was measure for the same. We chose general purpose computing platforms as a base to test the computational power as we are testing the system to work as a IOT Gateway Server.

At first we have created many threads using POSIX APIs and set priority the tasks .Later by using scheduler we assigned the tasks to the specific CPU cores.

For example if there are 10 tasks , and if tasks 2, 7 are assigned to CPU1 and 1,4,8,9,10 are assigned to CPU3 and tasks 3,5,6 are assigned to CPU4,Here CPU2 is left idle. Then the CPU 1,3,4 perform only related and internal tasks where as CPU2 performs only internal tasks, rest of the time it remains idle. Hence in this way we have implemented asymmetric processing on multi core and checked for CPU performance and measured the start time, end time and latency of all the tasks.

CHAPTER-8

CONCLUSION AND CONTRIBUTION TO THE SOCIETY

We implemented the synchronous processing on standard Linux kernel which is configured on Xenomai and performed functional tests like latency, clock and switch tests. And implemented asymmetric processing on various processors like i3, AMD, i5 processors and calculated the latency values and we have also verified the multi processing on system level using system monitor performance and graphs were plotted for various asymmetric processing on various processors.

CHAPTER-9

REFERENCES

[1] Nikola Markovic, Daniel Nemirovsky, Osman Unsal, Mateo Valero and Adrian Cristal "Kernel-to-User-Mode Transition-Aware Hardware Scheduling" IEEE Micro (Volume: 35, Issue: 4, July-Aug. 2015)pp. 0272-1732.

[2] M.Aater Suleman,Onur Mutlu, Moinuddin Qureshi, "Accerlating Critical section Execution with Asymmetric Multicore Architecture" IEEE Micro (Volume:30, " , Issue: 1 , Jan.-Feb. 2010).

[3] Junji Sakai, Inoue Hiroaki, Sunao Torii, Masato Edahiro. "Multitasking Parallel Method for High-End Embedded Appliances." IEEE Micro (Volume: 28, Issue: 5, Sept.-Oct. 2008) **pp:** 0272-1732.

[4] H. Roth, A. Chandra "POSIX standards for fault management in a real-time environment". IEEE Xplore**:** 06 August 2002 pp. 0-8186-7515-2.

[5] Emily Blem, Hadi Esmaeilzadeh, Renee St.Amant, Karthikeya Sankaralingam, Doug Burger, " MultiMore Model from Abstract Single Core Inputs". IEEE computer Architecture Letters (Volume: S12,issue 2 dec.2013); pp.1556-6056

[6] Yang Zhou, Qiaodi Zhou. " The embeded real-time Linux operation system based on the Xenomai." Electrical and Control Engineering (ICECE), 2011 International Conference on 16-18 Sept. 2011 Published on 24 October 2011

[7] M.D.Marieska, A.I.Kistijantoro, M.Subair." Analysis and benchmarking performance of Real Time Patch Linux and Xenomai in serving a real time application". Electrical Engineering and Informatics (ICEEI), 2011 International Conference on 17-19 July 2011 Published on 19 September 2011.

[8] http://ask.xmodulo.com/view-threads-process-Linux.html.

[9]http://marc.merlins.org/Linux/Linux.conf.au2001/Day1/threads.pdf

[10]http://www.cs.fsu.edu/~baker/realtime/restricted/notes/pthreads.html

[11] https://www.Linux.com/learn/how-install-and-configure-conky

[12] https://www.ibm.com/developerworks/library/l-async/

[13] http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html

[14] https://randu.org/tutorials/threads/#pthreads

[15] http://scitechconnect.elsevier.com/asymmetric-multi-processing-amp-vs-symmetric-multi-processing-smp/

[16] http://Xenomai.org/downloads/Xenomai/stable

[17] http://github.com/dankex/tools/tree/master/Linux-kernel/wake_latency/