I. ABSTRACT

Most of the embedded systems have real-time requirements about the use of Real-Time Operating Systems able of satisfying the embedded system requirements. So, in embedded application where the real-time operating systems are noted are been called as the Real Time Embedded Operating Systems (REOS). Main features of that of real-time systems would be discussed with this paper. To get a big picture we basically take a look at many structures and standards of REOS. Then, issues and techniques for different key topics which are been scheduling, synchronization, and memory management are been discussed. In this paper we are presenting the new way to tackle the problem with the resource reservation abstraction and an appropriate model for schedulability analysis. In this work, the architecture for a runtime CPU scheduler customization framework for the Linux kernel that take into account different applications requirements is presented.

Index Terms— Runtime Scheduling Policies; CPU scheduler customization;

II. INTRODUCTION

Timing conditions are given to the real time systems. Performance of the Real time system is been shown in the terms of the ability to take decisions in an appropriate manner. Deadlines for fulfillment is been provided for these problems. A skipped deadline would be as wrong as a bad answer. Real-time systems are divided as hard, firm or soft systems depending on the hardness of that of the harm caused by the skipped deadline. Real-time systems in which skipping a deadline is catastrophic are been called as hard real time systems. Real time systems are provided withCompact deadlines, but where the certain low
probability of skipping a deadline can be tolerated. If system does not allow deadline to be skipped at the times and still could be recovered, they would be called soft real time systems [16]. Computing systems could be called as embedded systems with tightly coupled hardware as well as software through which a function is been carried out. Specialized real-time computer system which is a part of big system.

Real time systems are provided with timing conditions. Real-time system’s performance is been provided in the terms its capability to take decisions in a regular manner. A missed deadline would be as wrong as a bad answer. Real-time systems are classified as hard, firm or soft systems depending on the hardness of the harm caused by a missed deadline. Real-time systems in which miss a deadline is catastrophic are called hard real time systems. Firm real-time systems have compact deadlines, but where a certain less probability of skipping a deadline can be tolerated. If deadlines are allowed by the system to be skipped at times and still could be recovered, they would be called soft real time systems [16].

Choice of a proper language is important in designing a RE system, task partitioning and merging, and giving priorities to handle response times. Merging highly cohesive parallel duties for sequential execution would reduce overheads of context switches and inter-task communications. High priorities should be provided complex tasks by the designers. To bypass starvation, care should be taken which occurs when higher priority tasks are ready to run, which would result in insufficient processor time for less priority tasks. Additionally, a small memory footprint is needed by good response time in resource-impoverished systems.

This paper shows how the problem can be tackled by using the resource reservation abstraction and an appropriate model for schedulability analysis. Along with the application increased and the load aggravated in Linux, the Linux scheduler couldn't meet the customer's need. A new tactic of scheduler is given after analyzing sonic insufficiency of old scheduler. The system performance test expresses that adoption new scheduler to have to promote vary greatly.

CPU scheduler is a very important subsystem which affects system throughput, interactivity and fairness. The development of Linux kernel is relatively fast-paced. By now, many CPU schedulers have been designed by researchers, hobbyists and kernel hackers. It is necessary to accurately compare and analyze different characteristics among these schedulers, so as to understand and design better CPU schedulers for various applications. However, researchers lack a straight-forward method to compare and analyze these CPU schedulers precisely.
We also analyze the impact of implementations of schedulers on fairness and interactivity of applications, discuss challenges in estimating application resource requirements in different environments, and present some ideas for developing future CPU schedulers. Developing CPU scheduling algorithms and understanding their impact in practice can be difficult and time consuming due to the need to modify and test operating system kernel code and measure the resulting performance on a consistent workload of real applications. To address this problem, we have developed WARP, a trace-driven virtualized scheduler execution environment that can dramatically simplify and speed the development of CPU schedulers. WARP is easy to use as it can run unmodified kernel scheduling code and can be used with standard user-space debugging and performance monitoring tools. It accomplishes this by virtualizing operating system and hardware events to decouple kernel scheduling code from its native operating system and hardware environment [10] [11].

III. RELATED WORK

Based on the system model introduced which only considers applications using one hardware resource (the CPU), the schedulability of a real-time task set (i.e., the fact that the tasks’ deadlines are respected) can be guaranteed by using a proper scheduling algorithm and an admission test. The admission test can be carried out in different ways (utilization-based test, response time analysis, or time demand analysis).

However, this model is simplistic because it does not consider the time consumed by the OS kernel to handle interrupts raised by the hardware devices connected to the system. For example, consider a traditional kernel, in which hardware interrupts are generally served in two phases: [2] [3]

- a short Interrupt Service Routine (ISR) is invoked as soon as an interrupt fires and is responsible for acquiring knowledge the hardware interrupt mechanism, postponing the real data transfer and processing to a longer routine, to be executed later;
- a longer routine (soft interrupt, or bottom half) is executed later to correctly manage the data coming from the hardware device.
- Both ISRs and soft interrupts have a higher priority than user tasks, and can “steal” execution time from them.

This implies that a low-priority task can affect the schedulability of high-priority tasks by causing the generation of a large number of hardware interrupts. [5]

IV. THE STRUCTURE OF REOS

RE system’s functionalities are traditionally abstracted into assignments, and these assignments undergo different states which are activated, suspended, waiting etc. These states are
mainly handled by hardware interrupts. In this section we review some popular models helped to build real-time software.

A. A Basic Interrupt-driven Task Model
This is the easiest model and mainly used to design simple real-time software. Here each system activity is seen as a task and the code for every task is written as an interrupt service routine (ISR). The interfaces to different devices are written as interfacing routines that could be used as ISRs. Then easy control routine is used to equalize their activities.

Hardware timers are helpful to generate needed interrupts during execution. When an interrupts is detected, it is handled by following steps.
1. The context of the running task is saved.
2. The interrupt is detected and its associated ISR is called.
3. The saved context is restored and the execution of the interrupted task is resumed.
ISRs could be preempted by high priority interrupts, but in some systems ISRs are non-preemptible. The basic feature of this model is that the tasks (ISRs) are handled carefully for better speed and predictability. Exokernel based design is a same approach of minimum operating system. In this approach the operating system should show only very vital support which could be allocating resources to tasks, protecting tasks from each other, revoking access to resources, etc., leaving the other of the high level policies to the application developers. The use of this approach is to improve performance, but it is difficult to write applications on exokernel-based systems.

B. The Nanokernel-Based Model
This approach is basic step towards giving software support for real-time application development and execution. The tasks are considered as regular and time-critical. Then, the time-critical tasks are handled by the interrupt-driven technique as discussed above and general tasks are handled by the kernel [12]. The kernel shows following set of services:

- Task creating and deleting
- Task scheduling;
- Timing and interrupt management.

At a conceptual level, this model consists a hardware abstraction layer for the devices. From the Figure 1, the kernel is in layer 2, the interfacing and interrupt software is in layer 3, and applications are in layer 3.

![Figure 1: Nano kernel Based Model: Layered View](image)
C. The Microkernel-Based Model

The microkernel model just provides the main mechanisms needed to implement the operating system policies. The basic for the microkernel was to implement the needed basic operating system primitives so the operating system services could be implemented on the top of the kernel. The embedded OS QNX and Sambaing are Micro kernel-based operating systems. Symbian has different layers with a nano-kernel for the innermost layer, the symbian OS kernel for the second level, then microkernel servers, and lastly user applications in the outermost layer.

D. The Monolithic-kernel-based Model

Any design not following the given philosophies could be considered under such a model. In this case, the OS is simply a piece of code containing different modules. One module calls the other in one or more than one way.

In this model, the complete operating system could be viewed as a monolithic structure as shown in Figure 2 and hence referred to as monolithic kernel based systems. RT-Linux is monolithic kernel based real-time operating system and Linux kernels are highly portable and simply configurable.

Figure 2: Monolithic structure of Operating System

Typically, added functionalities in this model are the following:

- Cultured CPU scheduling;

- Solutions for priority inversion issues;

- developed memory management;

- File handling;

- Graphics handling;

- Networking.
V. PROPOSED ARCHITECTURE

Mobile devices have been growing rapidly the last few years, at the same time the user requirements have increased considerably. The growing trend has been seen due to the real-time feature support of the recent Linux kernel. CPU scheduler relies on a particular scheduling algorithm (policy), for example FIFO, Round Robin and SJF. Each of these policies has different properties and the selection of a particular policy depends on specific criterion such as CPU utilization, throughputs, response time, deadlines-met, predictability, proportionality, and waiting time.

The advent of new mobile processors architectures such as multi-core CPU’s from Intel and cell processors, advanced scheduling policies are required to benefit from these architectures i.e. achieve a balance between power saving and good performance with multi-core mobile processors. Some advantages of using such framework to switch between the scheduling policies includes usage flexibility, simplify management and maintenance of the mobile operating system. RCSC will provide the flexibility to handle different requirement of the user and the applications. As the policies are pluggable during runtime, the mobile users, developers or the operating system itself could adapt to the changing requirement. The task of kernel developers to enhance and evaluate scheduling policies will be simplified as testing can happen during runtime.

CPU Scheduling Simulator:

CPU Scheduling Simulator (CPUSS) is a framework that allows you to quickly and easily design and gather metrics for custom CPU scheduling strategies.

- CPUSS records the following metrics about your scheduling algorithm:
  - Average process wait times
  - Idle CPU time
  - Busy CPU time
  - Wait time mean
  - Wait time standard deviation
  - Response time mean
  - Response time standard deviation
  - Turnaround time mean
  - Turnaround time standard deviation
  - Throughput stats
  - Throughput mean

- For each process
  - Arrival time
  - Start time
  - Completion time
  - CPU activity
  - Burst time
  - Id
  - Priority
  - Wait time
  - Turnaround time
  - Response time
As well as the core features CPU scheduling simulator also allows you to:

- Define the processes to schedule
- Auto generate the processes to schedule (varying burst time properties)
- Log results to SQL Server
- Hook into events
  1) Simulation Started/Completed
  2) Process Started/Preempted/Resumed.

The Algorithms which we use are:

- First Come First Served
- Round Robin (time quantum can be defined)
- Shortest Job First*
- Priority First*
- SJF with Priority Elevation rule (threshold can be defined).

**How Algorithm works**

The simulator uses the algorithms listed below:

- First-Come, First-Served (FCFS):
  Processes are assigned the CPU in the order they request it.
- Round-Robin (RR):
  Each process is given a limited amount of CPU time, called a time slice, to execute. If the required CPU burst of the process is less than or equal to the time slice, it releases the CPU voluntarily. Otherwise, the scheduler will preempt the running process after one time slice and put it at the back of the ready queue, then dispatch another process from the ready queue.

- Shortest-Job-First (SJF):
  When the CPU is available, it is allocated to the process that has the smallest next CPU burst.

- Shortest-Remaining-Time-First (SRTF):
  When the CPU is available, it is allocated to the process that has the shortest remaining CPU burst. When a process arrives at the ready queue, it may have a shorter remaining CPU burst than the currently running process. Accordingly the scheduler will preempt the currently running process.

### VI. WORK DONE

We implemented the proposed approach on Java (J2ME platform) for Mobile Operating System Scheduler. At first stage we carried out implementations on Windows platform. Following snapshots are showing the results for efficient REOS scheduling framework.

Figure 3 showing the generation of arrival time and burst time calculation:

Figure 4 showing the results for proposed mobile operating scheduler simulations such as average waiting time, average turnaround time and throughput etc. From those results we can say that these proposed results are more improved as compared to existing frameworks.
VII. WORK DONE
In these paper characteristics of RE systems, we reviewed. After this, for real time embedded operating systems we look at various structures and standards. Then we discussed issues related to memory management and scheduling in REOS. At last for each category of real-time applications we select prominent, popular REOS and discuss its features.

In this paper, a new runtime CPU scheduler customization framework is proposed due to the limitation of Linux CPU scheduler implementation. RCSC architecture provides pluggable scheduling policies and important feature for kernel developers and normal users to evaluate and customize the mobile operating system to suit the workload, applications and environment. Our simulation results showing that proposed algorithms having better throughput and minimum waiting time.

REFERENCES

Figure 3: Main Screen
Figure 4: Results from Simulations


VI-74 Volume 1

