International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I8P114
Volume 72 Issue 8, 98-103, August 2024
© 2024 Seventh Sense Research Group®

Original Article

A Practical Overview of Real-Time System Implementation in Linux Kernel

Shobhit Kukreti[1], Priyank Singh[2], Tanvi Hungund[3]

[1]*Carnegie Mellon University, PA, USA.*
[2]*Rochester Institute of Technology, NY, USA.*
[3]*Cal State Fullerton, CA, USA.*

[1]*Corresponding Author : skukreti@ieee.org*

Received: 17 June 2024 Revised: 21 July 2024 Accepted: 10 August 2024 Published: 30 August 2024

***Abstract** - Over the years, in the system software world, Real-Time Systems[1] have made a large impact in a variety of applications, including mission-critical environments. Such applications typically require a predictable response in a timely fashion to be successful. In areas like aerospace, medical robotics, industrial automation and telecommunications, real-time systems real-time systems have had the biggest impact. Recent trends like autonomous vehicles and smart grids facilitate seamless and reliable operations between components. As technology advances, Real-Time Systems have become a fundamental building block that is often hidden and mostly misunderstood. This paper intends to shed light on real-time systems while also adding an implementation of Rate Monotonic Systems[2] in Linux Kernel[3].*

***Keywords** - Linux kernel, Real-time system, Operating system, Rate monotonic scheduling, Task management.*

1. Introduction

A real-time system is a type of computer system which has strict timing constraints when executing it's task. Therefore, while the logical correctness of the output is of importance, its temporal correctness is of the same importance. For instance, an automotive braking action should be completed within a specific time period when the system applies the braking input. Broadly, real-time systems are of two types: hard and soft. Hard real-time systems result in catastrophic failure if the timing constraints are not met, as mentioned in the example above.

However, soft real-time systems, while still being time-sensitive, do not cause the same catastrophic failure. An example of a soft-real time system is an online video streaming application where if a certain video frame is missed, it may reduce the video quality.

The rest of the article is defined as follows:

Section II gives a summary overview of the real-time system. Section III demonstrates the Linux Driver development to add real-time characteristics to a non-real-time system. Section IV shall be the result and conclusion, followed by references.

2. Overview of Real-Time Systems

Fig 1. describes a typical operating system[4] stack. It has user applications, a library which abstracts away all the complex APIs exposed by the kernel.

In the same context, the real-time system software stack will look similar to the one shown in Figure 1, with certain optimization, extensions added to the Scheduler and Task Management.

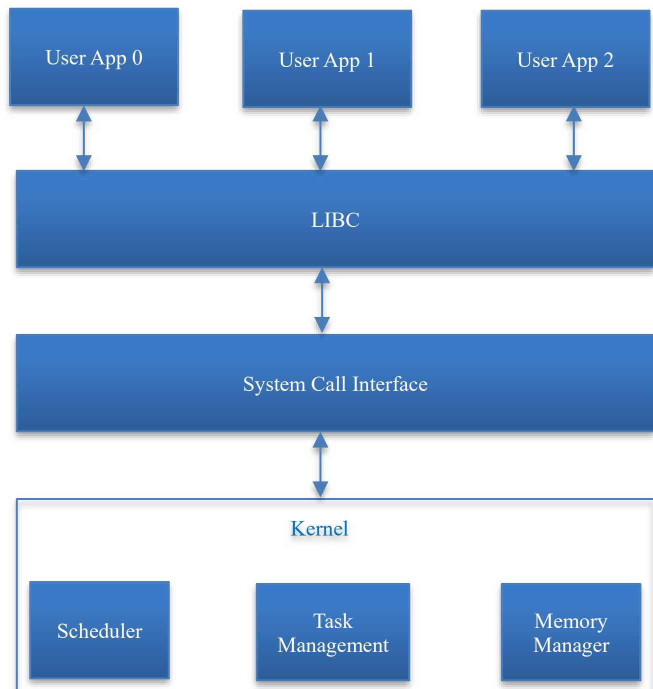

Fig. 1 Operating system

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

The user application requests access to a hardware resource via means of a System Call. System Call switches the operating system mode from, say, user to a Privileged mode where hardware resources can be accessed via device driver(s), amongst other operations of the kernel.

In a Real-Time Embedded System (RTES), the services it performs are well-defined and known prior to the deployment. This is somewhat necessary since the criticality of service needs to be known prior to deployment. Each service can be considered a software task (job).

Every task has a timing constraint. The timing constraint is specified in terms of the deadline before which the task needs to be completed. Based on the criticality of a task and whether the deadline can be missed, the RTES can fall into two categories: Hard or soft Real-time Time Systems.

2.1. Hard Real-Time Systems

In hard real-time systems, missing a deadline is unacceptable. These systems are found in environments where timing precision is critical, as in life-support systems or automotive airbag controls. Missing a deadline can lead to catastrophic failures, necessitating deterministic scheduling algorithms.

Characteristics**:**
Schedulability: The tasks should meet their timing constraints.
Low Latency: Deterministic worst-case response time.
Error Handling: Error handling to prevent system failure due to missed deadlines.

2.2. Soft Real-Time Systems

Soft real-time systems, while still time-sensitive, allow for some flexibility. Missing a deadline results in degraded performance rather than catastrophic failure. Systems such as online video streaming are deemed soft real-time.

Characteristics:
Adaptive Scheduling: Dynamic scheduling algorithms adjust based on workload, providing flexibility in meeting deadlines.
Graceful Degradation: Performance drops are handled in a way that allows the system to continue operating.
Resource Efficiency: These systems optimize resource usage to balance performance with timing constraints.

There are broadly two classes of algorithms, clock-driven and priority-based as shown in Figure 2.

Table 1. Clock driven algorithm

First Come, First Serve	Non-preemptive Tasks run in the order of arrival.
Round Robin	The task receives a fixed quantum of the CPU to run.

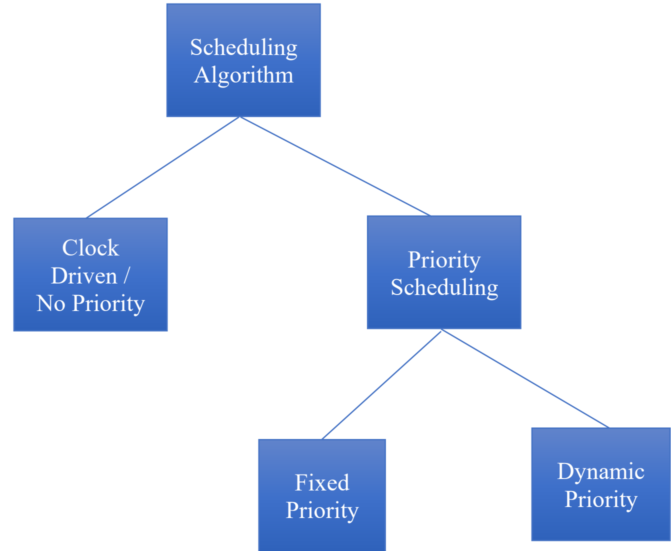

Fig 2. Scheduling algorithms

Table 2. Fixed priority preemptive algorithm

Rate Monotonic Scheduling	Priorities are assigned based on the rate of the periodic task. The lower the time period, the higher the priority.
Deadline Monotonic Scheduling	Tasks are assigned priority based on deadline. Shorter deadlines have a higher priority.
Shortest Job Scheduling	Task priority is based on the run-time of the process. Lower run-time receives higher priority.
Multi-Queue Round Robin	Multiple queues with different time quanta. Tasks are added to a queue based on priority.

Table 3. Dynamic priority preemptive algorithm

Earliest Deadline First	The task is assigned priority based on a deadline. Shorter deadlines have higher priority. Priority changes on the fly.

Utilization (U) = Compute Time (C) / Time Period (T).

The sum of utilization of all tasks should be less than 100%.

2.3. Rate Monotonic Scheduling

It is an optimal fixed scheduling policy, i.e. if a set of tasks cannot be scheduled by Rate Monotonic policy, then it may not be schedulable by any other fixed priority algorithms.

It is proven by Liu and Layland[5] that under Rate Monotonic policy, a set of tasks is only schedulable if their utilization is up to 69.3%.

Table 4. Number of Tasks vs Utilization (Upper Bound Test)

Num Tasks	Utilization
1	1
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
…..	
∞	0.693

Table 5. Sample task set

Tasks	C	T	U
T1	4	10	0.4
T2	4	15	0.27
T3	10	35	0.29

T1 has higher priority than T2, T2 has higher priority than T3 based on their time period (T)

Here, a set of T1 and T2 together are schedulable as utilization of 0.67 < 0.828 from Table 4.

T1, T2, and T3 all together have a utilization of 0.96, which is greater than 0.780 when the number of tasks (n) = 3. Hence, the task set is not schedulable under the upper bound test.

Response Time (RT) test has to be applied when a set is not schedulable under the Upper Bound test. However, we will use the upper bound test when implementing the RMS policy in Linux.

3. Implementation

Several CPU scheduling policies are supported in a modern kernel, each tailored to manage particular kinds of workloads effectively. In Linux systems where real-time processes are not required, the Completely Fair Scheduler (CFS) predominantly handles the CPU scheduling tasks. CFS is a widely used scheduler. It consists of sophisticated heuristics that aim to optimize performance across a diverse array of workloads. CFS offers various adjustable parameters to fine-tune its policy. Despite its complexity and adaptability, CFS may not satisfy all use cases which is a testament to the challenges in CPU scheduling.

In this section, we will discuss steps about implementing a real-time system by extending the Linux kernel. To put it succinctly, at the implementation level, an RTS requires task management based on the parameters (C,T) set by the system architect. The values of C,T are system-dependent and requirement-dependent.

3.1. Task Management

A task can be in one of the multiple states on its inception, such as Ready, Running, Waiting and Exiting. A new task when created, is added to the Ready Queue. When the scheduler tick/irq occurs, the scheduler picks the next task to run. In our RTS system, recall that we shall pick the state based on the higher rate of occurrence as a higher priority.

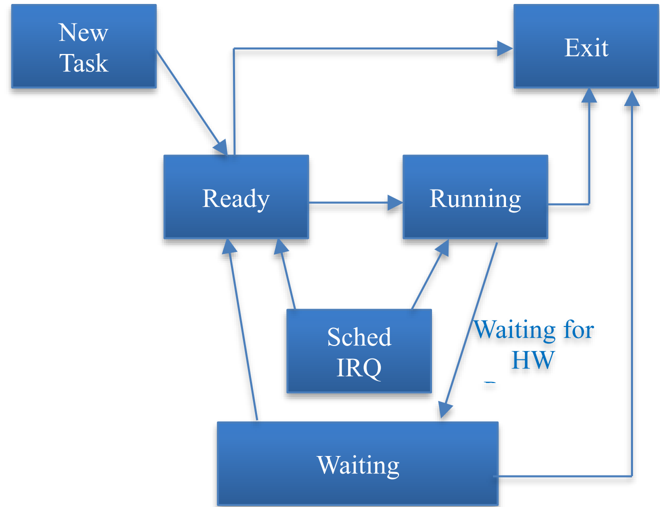

Fig. 3 Life of a Task in OS

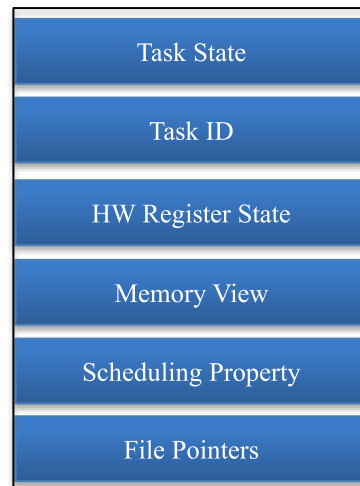

Fig. 4 Task control block

```
struct task_struct {
unsigned int__state;
unsigned intflags;
....
intprio;
….
struct pid *thread_pid;

/* Monotonic time in nsecs: */
u64 start_time;
.....
};
```

Fig. 5 Linux TCB

Every task is described with a Task Control Block (TCB). A TCB describes the task's current state, its scheduling attributes, open file references, hardware register state, and the memory it can access. On Linux Kernel which is our target platform for implementation, the TCB is defined with a struct task_struct. Figure 5 shows the task_struct with a few attributes displayed.

Based on the information we have about the task state and TCB, task_struct from Linux, we will add our own parallel OS within the kernel space.

1. We add a new system call for our driver. This is required by the user-space application to specify three parameters.

- Process ID or PID
- Worst Case Execution Time or 'C' in msec
- Time Period or 'T' in msec

2. The linux kernel calls a function called ***context_switch (task_struct *old, task_struct *new)*** when it switches between tasks. We add a hook here which calls a function called ***force_rt()*** in our driver.

3. Setup code for Linux High-Resolution Timer[6]. We start an hour timer with a tick of 10 microseconds (configurable).

4. Implement the RMS Upper Bound Algorithm with a partitioned scheduler scheme in which task sets are grouped together to run on certain cores of the CPU.

5. Add sysfs entries show() to show our driver's internal state.

```
struct rt_sched_queue {
struct task_struct *tsk;
struct rt_sched_queue *next, *prev, *cpuq;
int utils;
// capture actual time spent
struct timespec cts, pts, ets, rawts, rawets, rawpts;
spinlock_t lock;
int state, ctimer_state, ptimer_state, raised_exit;
// specified by user
int e_time, c_time, p_time;
pid_t pid;
struct kobject pid_kobj;
….
};
```

Fig. 6 RTS driver data structure

Figure 6 shows the design of the driver queue data structure. It holds the process ID, user-specified timing constraints, actual time spent by the task running on the system and some attributes required for the Linux ***sysfs*** framework.

```
asmlinkage long sys_rt(pid_t procid, int comp_time, int
period)
{
sched_queue *tmp;
// check if dup exit for the pid
tmp = check_dup(&runqueue, (pid_t)procid);
if (tmp !=NULL) return -1;

tmp = create_node();

tmp->pid = procid;
tmp->c_time = comp_time;
tmp->p_time = period;

tmp->tmr_c.timer_callback = &c_callback;
tmp->state = TSK_RUN;

tmp->ctimer_state = CLK_STOP;
tmp->ptimer_state = CLK_STOP;

struct task_struct *tsk =
                          find_task_by_vpid(procid);

// raise the priority of our task
tsk->prio = 105;
// Set the scheduling policy as Round Robin
tsk->policy = SCHED_RR;

/* Insert Node in Per processor Queue */

// Worst Fit Descreasing Packaging per Core
if (wfd_packing() == -1) {
    delete_node(procid);
    return -1;
}
// create sysfs entry for user view
create_pid_sysfs(tmp);

// Assign task set per CPU Core RunQueue
if (assign_task_core(cpu0, 4) != 0)
pr_err("new Task Set Not Schedulable");

if (assign_task_core(cpu1, 5) != -1)
pr_err("new Task Set Not Schedulable");
if (assign_task_core(cpu2, 6) != -1)
pr_err("new Task Set Not Schedulable");
if (assign_task_core(cpu3, 7) != -1)
pr_err("new Task Set Not Schedulable");
...
}
```

Fig. 7 Code snippet of the system call

Our Linux HR Timer callback forms the basis of the system tick of our implementation. In the ***hr_timer_callback***

we iterate over all the task(s) in our queue advance the running ‘C’ and ‘T’ values. Once a task has finished its ‘C’ execution units, we call the per task assigned callbacks to move the Task into a TASK_SUSPENDED state. Similarly, when a task finishes overall of ‘T’ units, it is woken up and added back to the ready queue.

```
void os_callback(void)
{
sched_queue *node = runqueue;

while (node != NULL) {
  if (node->tmr_c.fired == true) {
    node->tmr_c.timer_callback(node);
    node->tmr_c.fired = false;
    node->tmr_c.active = false;
  }
  if (node->tmr_p.fired == true) {
    node->tmr_p.timer_callback(node);
    node->tmr_p.fired = false;
  }
node = node->next;
}
}
```

Fig. 8 Main call back of HR timer

We use bin packing heuristics to pack the task in a set which will run on a specific core. By setting the CPU affinity of a task, we bind the task to a core. Recall that your upper bound test means we can only load our code to 69.3% utilization to achieve RMS.

In our code implementation, we pick the Worst Fit Heuristic. The Worst Fit Decreasing (WFD) heuristic is a strategy used in bin packing problems to minimize the number of bins required. This heuristic involves two primary steps: sorting the items in descending order based on size and then applying the Worst Fit method. In the Worst Fit method, each item is placed in the bin with the most remaining space that can still accommodate the item. By prioritizing the emptiest bins, the WFD heuristic tends to distribute items more evenly across the available bins compared to other methods. This forms the basis of our simple algorithm of sorting the tasks and then placing them in sets (buckets) assigned to a core.

4. Result andConclusion

With our algorithm in place and the driver compiled into the kernel. We create some user applications which we would want to adhere to the RMS policy.

Figure 9 shows the internal state of our driver, which is managing tasks identified by the PID (P).

C_ms and T_ms are the user-specified real-time constraints. RC and RT values indicate the actual time spent by the task in the kernel either running, waiting or suspended. ‘pre’ number tells us how many times the kernel switched out the task. The kernel exports RC and RT values to user space via sysfs. The above result indicates that:

$$(C/T) == (RC/RT)$$

```
root@exynos8890:/sys/rtexynos# cat 2684/proc_info
P:2684, RC:29580.065102891 RT:88789.407255030,        C_ms:50, T_ms:150, pre:263344
root@exynos8890:/sys/rtexynos# cat 2676/proc_info
P:2676, RC:11105.961453599 RT:88841.972716273,        C_ms:40, T_ms:320, pre:381091
root@exynos8890:/sys/rtexynos# cat 2674/proc_info
P:2674, RC:18368.211189501 RT:88878.361290238,        C_ms:20, T_ms:100, pre:138465
root@exynos8890:/sys/rtexynos# cat 2672/proc_info
P:2672, RC:22530.296558111 RT:88934.235933284,        C_ms:50, T_ms:200, pre:553779
root@exynos8890:/sys/rtexynos# cat 2677/proc_info
P:2677, RC:19071.738699219 RT:88939.426235797,        C_ms:10, T_ms:50, pre:135878
root@exynos8890:/sys/rtexynos# cat 2675/proc_info
P:2675, RC:22545.533814101 RT:88982.724305886,        C_ms:30, T_ms:120, pre:587137
root@exynos8890:/sys/rtexynos# cat 2683/proc_info
P:2683, RC:18136.279283799 RT:88982.824528994,        C_ms:30, T_ms:150, pre:283833
root@exynos8890:/sys/rtexynos# cat 2673/proc_info
P:2673, RC:22564.722848910 RT:89050.682263820,        C_ms:40, T_ms:160, pre:598180
root@exynos8890:/sys/rtexynos#
```

Fig. 9 Result

It means that the utilization which the user specified is what our real time driver was able to achieve for the user-task. While our implementation worked, we made some assumptions about no tasks accessing a shared resource. If there is a lock shared across two tasks with both of them accessing the shared resource, if the lower priority task acquires the mutex, then it leads to priority inversion.

In our case, it will lead to the violation of the timing constraints. There are locking protocols, such as Priority Inheritance Protocol and Highest Locker Protocol, but that is outside the scope of our current implementation and will be considered in the next implementation.

Our work is inspired from Linux/RK work done at Carnegie Mellon University[7].

References

[1] Real-Time Systems Overview, “Discover the Impact Real-Time Systems Have on Internet of Things Applications in Industries Ranging from Manufacturing to Healthcare to Oil and Gas and Robotics,” Intel, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/robotics/real-time-systems.html

[2] J. Santos, and J. Orozco, “Rate Monotonic Scheduling in Hard Real-Time Systems,” Information Processing Letters, vol. 48, no. 1, pp. 39-45, 1993. [CrossRef] [Google Scholar] [Publisher Link]

[3] The Linux Kernel Archives, Kernel, 2024. [Online]. Available: www.kernel.org

[4] Norman F. Schneidewind, *Operating Systems*, Wiley-IEEE Press, pp. 286-302, 2012. [CrossRef] [Publisher Link]

[5] C. L. Liu, and James W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 41-61, 1973. [CrossRef] [Google Scholar] [Publisher Link]

[6] Hrtimers - Subsystem for High-Resolution Kernel Timers, The Linux kernel, 2024. [Online]. Available: https://docs.kernel.org/timers/hrtimers.html
[7] Linux/Rk, Real Time and Multimedia Systems Lab, 2024. [Online]. Available: http://www.cs.cmu.edu/~rtml/