Computer Systems Department
Faculty of Computer and Information Sciences
Ain Shams University

Developing a scheduling framework for real-time operating systems

Thesis submitted as a partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences

By

Hesham Hussien Abbas Hussien

Teaching Assistant at Computer Systems Department,

Faculty of Computer and Information Sciences,

Ain Shams University

Under Supervision of

Prof. Dr. Said ghoniemy

Professor of Computer Systems

Computer Systems Department

Faculty of Computer and Information Sciences,

Ain Shams University

Prof. Dr. Eman Shaaban

Professor of Computer Systems

Computer Systems Department

Faculty of Computer and Information Sciences,

Ain Shams University

June – 2019
Cairo

Acknowledgment

First of all, I am deeply grateful to my supervisors: Prof. Said ghoniemy and Prof. Eman Shaaban for supporting and believing in me throughout the thesis process. I would like to thank my wife and my little daughter for their patience during this period. Finally, and most importantly, I would like to thank my parents for supporting and encouraging me to keep going on in my study.

Abstract

The complexity of embedded real-time systems has increased, and is expected to handle growing number of diverse applications. Most of these applications have large diversity in execution times of their tasks. Traditional scheduling techniques, such as Fixed-Priority Scheduling, Earliest Deadline First Scheduling (EDF), Rate Monotonic Scheduling (RMS), etc., do not satisfy the requirements of such applications. In most traditional scheduling techniques, one or group of tasks may dominate the CPU resources regardless of its criticality, which is called monopolism. In this context, the timing requirements of each application in the system should be isolated and guaranteed.

The Hierarchical Scheduling Framework (HSF) is an efficient solution for scheduling tasks of complex real-time systems. To avoid the interference between independent subsystems (applications) and guarantee a budget for each processor without preemption, HSF supports the concept of temporal isolation where each subsystem executes only in its server (virtual partitioning period).

This thesis proposed, designed and implemented an Adaptive Hierarchical Scheduling Framework based on EDF scheduler (AHSF-EDF), which creates and guarantees a virtual temporal isolation for each subsystem, thus resolving the problem of monopolization. AHSF-EDF implemented an adaptive budget controller that automatically and periodically adjusts the budget of each subsystem, to reserve resource wasting, based on Chebyshev’s estimator; a prediction algorithm for tasks execution times. Implemented into the kernel of TI-RTOS on a resource constrained platform, experiments show that the proposed scheme provides good performance for different applications with dynamic tasks under normal and overload conditions.

As an enhanced version of AHSF-EDF, we proposed an Adaptive Hierarchical Scheduling Framework based on EDF with Virtual Deadline (AHSF-VD) that dynamically adjusts the CPU budget of each server. Different relative deadlines are assigned to tasks depending on their criticality modes. Dual-criticality levels (low-criticality LO and high-criticality HI) are considered, where the virtual relative deadlines for high-criticality tasks are generated by greedy tuning algorithm. The proposed AHSF-VD framework implemented into the kernel of TI-RTOS, and tested in a real platform, is found to guarantee the minimum budgets for high-criticality servers during overload periods, and ensure that high-criticality tasks meet their deadlines with no miss ratios at the expense of low-criticality tasks.

List of Publications

- Hesham Hussien, Eman Shaaban and Said Ghoniemy. "Adaptive Hierarchical Scheduling Framework for TiRTOS". In The International Journal of Embedded and Real-Time Communication Systems (IJERTCS), Volume 10, Issue 1, Article 7, 2019. 121117-045223.

- Hussien, H., Shaaban, E., & Ghonaimy, S. (2018, December). Mixed-criticality Hierarchical Scheduling for TI-RTOS. In 2018 13th International Conference on Computer Engineering and Systems (ICCES) (pp. 279-283). IEEE.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
AHSF	Adaptive Hierarchical Scheduling Framework
AHSF-EDF	Adaptive Hierarchical Scheduling Framework based on EDF
AHSF-VD	Adaptive Hierarchical Scheduling Framework based on EDF with Virtual Deadline
A_i	Task's Arrival Time
AI	Artificial-Intelligence
AUTOSAR	AUTomotive Open System ARchitecture
B_s	Server's Budget
CAN	Controller Area Network
CCS	Code Composer Studio
CPU	Central Processing Unit
CT	Current Time
D_i	Task's Deadline
D_s	Server's Deadline
DAG	Directed Acyclic Graph
DBF	Demand Bound Function
DH_i	Task High-Deadline
DL_i	Task Low-Deadline
DM	Deadline Monotonic
DOM	Document object model
DWI	Decrease, Wait and Increase Protocol
E_i	Task's Execution Time
E'_i	Task's Remaining Execution Time
EDF	Earliest Deadline First Scheduling
EH_i	Upper Bound of Task Execution Time
EL_i	Lower Bound of Task Execution Time

FPU	Floating-Point Unit
GA	Genetic algorithm
GPIO	General Purpose Input Output
GPOS	General Purpose Operating System
HSF	Hierarchical Scheduling Framework
Hwi	Hardware Interrupt
ICSR	Interrupt Control and State Register
IMA	Integrated Modular Avionics
ISR	Interrupt Service Routine
$\mathbf{L_i}$	Task Laxity
LCM	Least Common Multiple
LLF	Least Laxity First
MCs	Mixed-criticality systems
$\mathbf{M_S}$	Sever Mode
MSP	Main Stack Pointer
MPU	Memory Protection Unit
NVIC	Nested Vectored Interrupt Controller
PID	(Proportional–Integral–Derivative) Controller
PSP	Process Stack Pointer
PWM	Pulse Width Modulation
RMS	Rate Monotonic Scheduling
RMS	Rate Monotonic Scheduling
RTOS	Real-Time Operating System
RTSC	Real Time Software Components
SBF	Supply Bound Function
SCB	System Control Block
SCR	System Control Register
Swi	Software Interrupt
$\mathbf{T_i}$	Task's Periodic Time
$\mathbf{T_S}$	Server's Periodic Time

T_s^{ctrl}	Server Control Period
TI-RTOS	Texas Instrument Real Time Operating System
U	Utilization
UIA	Unified Instrumentation Architecture
VTOR	Vector Table Offset Register
W	Sever Window
W_i	Tasks Weight
WCET	Worst-Case Execution Times
WIC	Wake-Up-Controller

Chapter 1
Introduction

This chapter presents an overview of Real-Time Operating Systems (RTOS) including their main features, classifications, and real-time task states. It also declares the thesis problem statement, motivation, objective and outlines.

1.1 Overview

An embedded system is a combination of hardware and software, designed for achieving specific functions, within relatively large systems. It is embedded as a part of a complete device.

In modern days, we can observe the spread of embedded systems around us, which makes our life safer and more comfortable. These embedded systems are almost ubiquitous and can be found in vehicles, planes, heart monitor watches, medical devices, smartphones, digital homes, Industrial equipment, agricultural systems, process industry devices, etc. The brain of these systems is the operating system which can be a General-Purpose Operating System (GPOS) or a Real Time Operating System (RTOS).

A GPOS is an unpredictable system, where the scheduler usually uses justice techniques to schedule its tasks onto the CPU to achieve high throughput, and does not guarantee that a high-criticality task will execute at the expense of low-criticality tasks. The throughput here means the total number of tasks that can accomplish their execution times or their work. On the other hand, a RTOS is mostly predictable system that uses priority preemptive scheduling techniques to schedule its tasks to meet their deadlines in time critical systems. Mostly in RTOS, a high-criticality task gets executed over the low-criticality tasks.

A GPOS is designed for high end devices such as PCs and server systems, etc. On the other hand, a RTOS is always targeting the stand-alone small devices such as Medical equipment, ATMs, etc. A RTOS has light weight for being suitable for such devices that hold low hardware configurations (RAM, ROM, CPU, etc.). A GPOS has an unbounded dispatch latency (mostly, when scheduling a large number of tasks) which is undesired; but, a RTOS can achieve bounded dispatch latency.

The primary GPOSs in use are Windows Server, Windows XP, 7, 8 and 10, IBM, macOS, as well as many versions of Linux and Unix. Some of the most widely used RTOSs are: TI-RTOS, FreeRTOS, VxWorks, QNX, eCos, RTLinux, etc. Moreover, most of current embedded systems depend on RTOS. The interaction between RTOS and other Layers is shown in Fig.1.

The features of RTOS, like multitasking, preemption, reliability, predictability, etc., make RTOS one of the most important pillars in embedded systems, especially with complicated systems that expect to accomplish their critical tasks before pre-set deadline.

Multitasking is the ability of the operating system to handle multiple tasks' operations within set deadlines, while the preemption is one of the key features in RTOS, where a critical task can preempt less-critical executable tasks to execute its operations.

Fig. 1. The interaction between RTOS and other Layers

For example, in case of Self-driving systems where the current running task is a regular task, like reading windshield wipers sensors, and suddenly an obstacle appeared on the route, RTOS should preempt the less critical current task and execute the critical tasks that handle this event, like executing vehicle's brakes and speed tasks.

Reliability is a very important feature of RTOSs, which keeps the system stable for 24/7 without human intervention; RTOS can take the right decision at the right time.

Predictability is a different key-feature of RTOS, where a system must perform its operations in a pre-defined time slots, respond in a predictable way to forecasted events, and produce the expected results. It is one of the most important features of RTOS to have a predictable (deterministic) pattern.

RTOS struggles to maximize the utilization of system resources with no failures. Processor modes include unprivileged and privileged modes. In user (unprivileged) mode, the software has a limited access to memory, CPU, and other system resources, while in privileged mode, the software has access to all system resources. RTOSs should have security modes when performing their instructions.

1.2 RTOS Classification

Real-time systems, that manage a group of time dependent peripherals (process the input data and give output in a given time), can be classified into three types (as shown in Fig. 2).

- **Hard Real-Time Systems**: strictly adhere to the task's deadline or limits of the task stipulated. Missing on a deadline can have catastrophic effects or a loss of life or property. For example, Air France Flight No. 447 crashed into the ocean when wrong readings of sensor caused a series of system errors. The pilots stalled the aircraft while responding to outdated instrument readings. All passengers and crew were killed.

- **Firm Real-Time Systems**: There is no value for a response that occurs past a specific deadline. Failure to meet the timing requirements is undesirable. Missing a deadline may not cause a catastrophic effect; but it could cause undesired effects. For example, in Satellite communication for monitoring some strategy targets, if one of the sending frames drop or delayed, it will not affect the whole transmission event.

- **Soft Real-Time Systems**: Missing a deadline is passable, as in streaming audio-video.

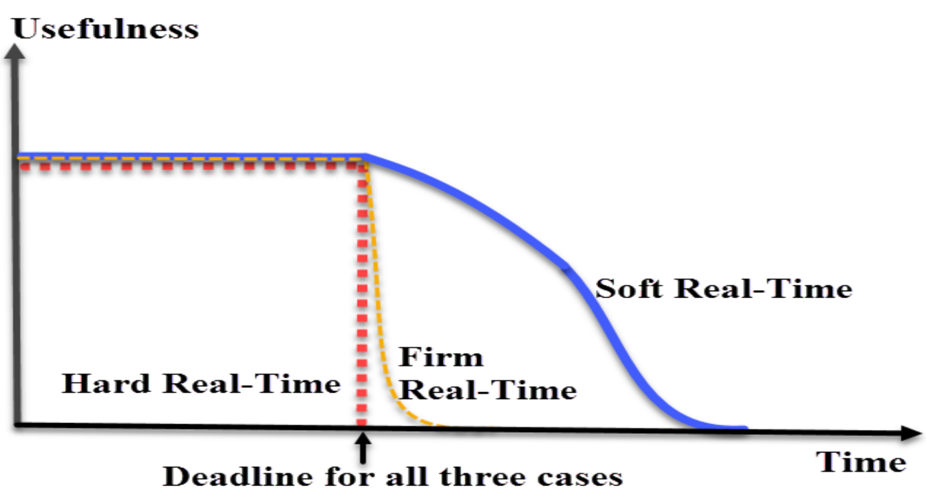

Fig. 2. Classification of RTOS (Hard, Firm and Soft) [1]

1.3 Task States

Each task's stack has its own context which represents its main parameters including periodic time, priority, deadline, stack size, current value of the program counter, and current state of the executing operations in CPU registers. Fig. 3 shows an example where the kernel decides to replace the running task 1 by task 2. Firstly, the kernel saves task 1 context from the CPU registers into its stack, then loads task 2 context from its stack into the CPU registers; this process is called context switching and is performed by the kernel dispatcher.

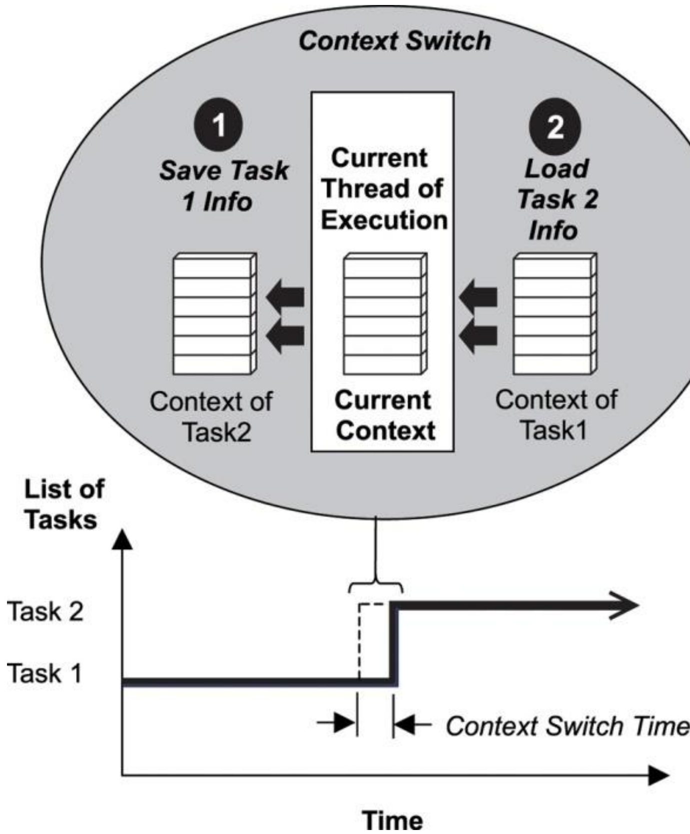

Fig. 3. Context Switch [2]

The Dispatcher plays its role after the scheduler selects the task to be executed. It allocates the CPU to the task through the following operations: switching context, switching to user security mode, and then jumping to the proper location in the user program.

A task can be created by allocating its stack size, and will be assigned to "Inactive" state until the admission of the scheduler (some systems permit it automatically) for switching to ready state as shown in Fig. 4. In the ready state, a task should complete the whole preparations for executing; however, it cannot execute if there is a higher priority task in the running state (running state is the only state where the task can use the CPU) which precedes it.

A task is ready to run and switches to the running state when no task precedes it. At this moment the dispatcher moves it to be executed by CPU. Once a task completes its work, it will be terminated (for non-periodic task) or returned to the ready state (for periodic tasks). However, a running task may be preempted for a while and switched back to the ready state in one of the following cases:

- Preempted by a higher priority task, or
- Interrupted by an interrupt service routine (ISR).

A RTOS has two approaches when the interrupt is over, either it returns to the Interrupted task (basic approach) or selects the current highest priority task (smart approach).

Also, a running task may be blocked for a while and switched to the waiting state in one of the following events:

- Waiting for a resource or I/O device to release, or
- Waiting an action from another task (in case of dependency-tasks).

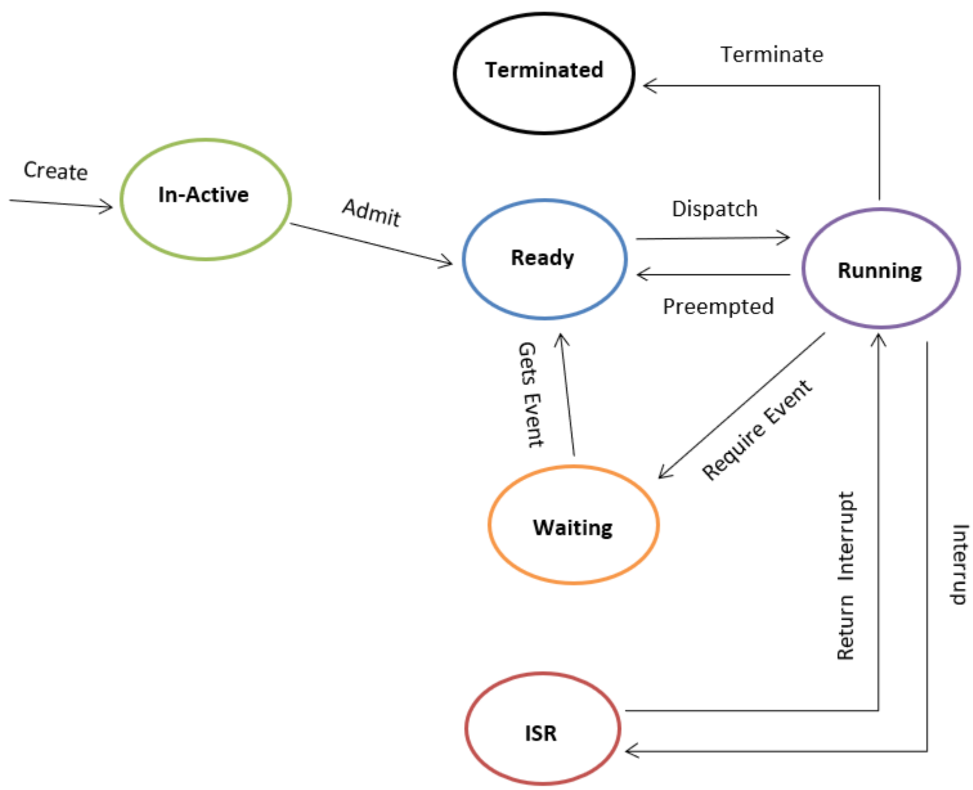

Fig. 4. Task States