Unit 4
Real-Time Operating System (RTOS)-II

Structure of the Unit

4.1 Unit Outcomes

After the successful completion of this unit, the student will be able to:

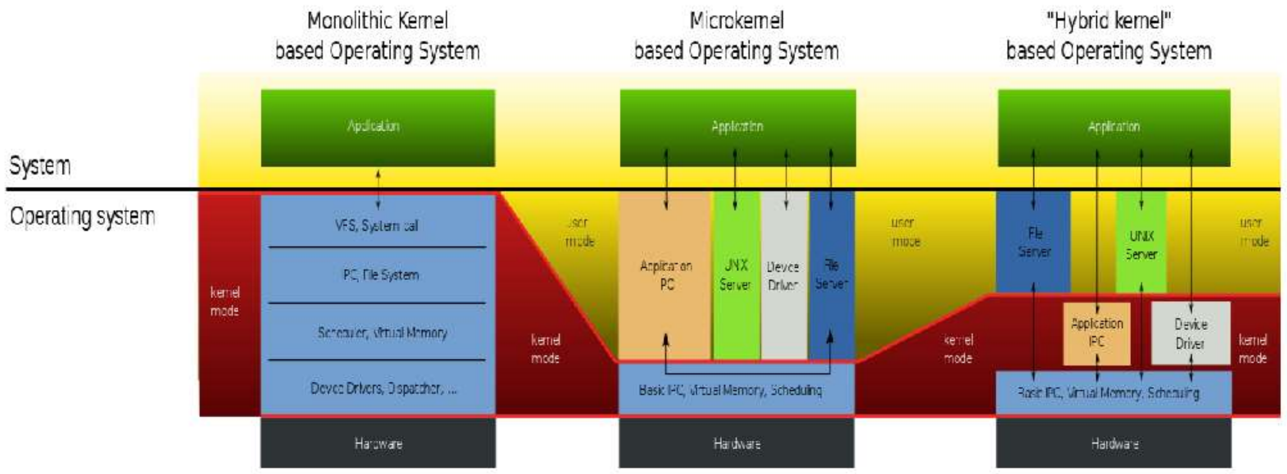
1. Define Real time operating system Kernel
2. Analyze scheduling algorithms for real-time systems.
3. Illustrate RTOS concepts with example RTOS.

4.2 RTOS kernel and function

In the realm of Real-Time Operating Systems (RTOS), the architecture evolves based on the complexity of the applications it supports. Initially, for simpler applications, an RTOS primarily consists of a kernel. However, as applications become more intricate, additional modules such as networking protocol stacks, debugging utilities, and device I/O functionalities are integrated alongside the kernel. This progressive RTOS architecture encompasses various components to meet the demands of diverse applications.

At the core of an RTOS architecture lies the kernel, which serves as a crucial intermediary layer between the underlying hardware and the running applications. The RTOS kernel encapsulates the intricate details of hardware interactions, shielding application developers from the complexities of hardware management. This separation allows developers to focus on application logic without needing to delve deeply into hardware intricacies.

Broadly, there are three distinct categories of RTOS kernels, each with its own characteristics and strengths:



Monolithic Kernel:

Monolithic kernels are integral components of Unix-like operating systems, such as Linux and FreeBSD. Essentially, a monolithic kernel is a singular program encompassing all the code essential for executing various kernel-related tasks. It manages fundamental system services, including process and memory management, interrupt handling, I/O communication, and file systems. By incorporating powerful hardware abstractions, the monolithic kernel reduces the frequency of context switches and inter-process messaging, resulting in enhanced performance compared to microkernels.

Microkernel:

In contrast, the microkernel approach involves a minimalist strategy. Microkernels focus on rudimentary process communication through messaging and control over I/O operations. Rather than handling hardware processes directly, a microkernel's primary functions involve memory protection management, inter-process communication, and process oversight. This streamlined approach to hardware abstraction leads to increased stability. Even if certain components, like file systems, encounter failures, the core microkernel remains unaffected. Operating systems like AIX, BeOS, Mach, Mac OS X, MINIX, and QNX adopt microkernels for their architecture.

Hybrid Kernel:

Hybrid kernels represent a blend of microkernel and monolithic kernel features. They maintain the simplicity and stability of microkernels while incorporating additional code into the kernel space. This allows specific code to operate more efficiently than if placed in user space. Hybrid kernels share traits with microkernels, yet they introduce certain elements associated with monolithic kernels, leading to a balanced architecture. Examples of operating systems adopting hybrid kernels include Microsoft Windows NT, 2000, XP, and DragonFly BSD.

The architecture of an RTOS, particularly concerning its kernel, adapts to the complexity of applications. It encompasses various types of kernels, including monolithic, microkernel, and hybrid each catering to different requirements and trade-offs in terms of performance, stability, and hardware control.

4.3 Task Management

A process represents a program that is currently running on a computer. It encompasses all aspects of program execution, including code, data, and the program's status. Each process operates in its own isolated memory space. Processes are relatively heavyweight in terms of resource utilization. A thread is often described as a "lightweight" process because it shares the same memory space with other threads within the same process. Threads within a process can efficiently communicate and share data since they all have access to the same memory and resources. Threads are known for their shorter creation and context-switch times compared to processes. Inter-Thread communication is generally faster and less resource-intensive than inter-process communication.

In the context of Real-Time Operating Systems (RTOS), the term "task" is synonymous with a "thread" or "lightweight process." A task, like a thread, represents a sequential program in execution within an RTOS environment. Tasks within an RTOS can communicate with one another and can also utilize shared system resources, such as memory blocks. Real-time tasks often come with timing constraints, meaning they must meet specific deadlines for their execution.

In Real-Time Operating Systems (RTOS), tasks are the smallest units of work. Applications are dissected into small, manageable, and sequentially executable units known as "tasks." They represent specific actions or computations that need to be performed. Tasks are created to achieve a particular goal, and they encapsulate a piece of code that accomplishes that goal. These tasks serve as the fundamental units of execution, and their behavior is regulated by three critical time-related properties: release time, deadline, and execution time. Efficient task management is crucial for handling time-critical operations.

Each task in an RTOS is governed by three fundamental time-critical properties:

1. *Release Time*: This property indicates the point in time when a task becomes eligible for execution. It's the earliest time at which the task can start running.

2. *Deadline*: The deadline is the point in time by which a task must complete its execution. Meeting deadlines is often crucial in real-time systems where timely responses are required.

3. *Execution Time*: Execution time is the amount of time it takes for a task to finish its work. This property directly impacts the system's ability to meet deadlines.

4.3.1 Task States:

Tasks can exist in various states within an RTOS:

- *Dormant:* In this state, a task is inactive and doesn't require any CPU time. The task has been created, and memory has been allocated to its structure. However, it is not in a ready state and cannot be scheduled by the kernel.

- *Ready:* Tasks in the "Ready" state are prepared to execute, but they are waiting for their turn to use the CPU. They are waiting for the scheduler to allocate CPU time. It remains in this state when another higher-priority task is scheduled to run and has control of the system resources.

- *Active:* When a task is in the "Active" state, it is actively running and utilizing CPU resources to perform its designated operations. It continues to run until it requires input (Inter Process Communication), waits for an event, or is preempted by a higher-priority task.

- *Suspended:* A task may be temporarily suspended or put on hold. In this state, its execution is paused, but it can be later resumed.

- *Pending:* Tasks in the "Pending" state are waiting for specific resources or conditions to become available before they can proceed with their execution.

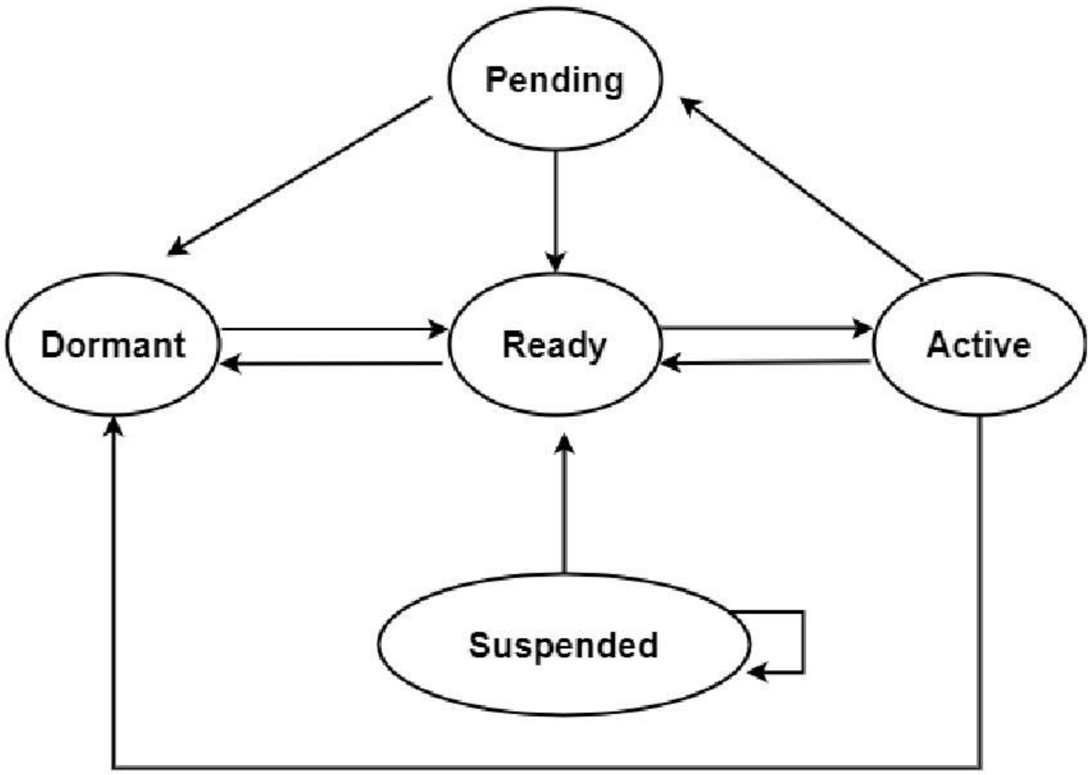


Figure 4.1:Task States

Throughout the execution of an application program, individual tasks continuously transition between these states. However, at any given moment, only one task is actively running, meaning it has control over the CPU. The process of switching CPU control from one task to another involves saving the context of the task about to be suspended while retrieving the context of the task about to be executed. This process is known as "context switching" and is pivotal in maintaining task scheduling and execution. During context switching:

- The context of the task about to be suspended (i.e., its registers, program counter, and stack) is saved.
- The context of the task about to be executed is restored.

This allows for seamless transitions between tasks while preserving their execution states.

4.3.2 Components of a Task:

A task in an RTOS typically comprises several key components:

Task Control Block (TCB): TCBs are data structures residing in RAM and are accessible only by the RTOS. They are used by tasks to remember their context, allowing for seamless context switching. They store information about the task, including its state, priority, and execution context. TCBs are used by the RTOS to manage and control tasks.

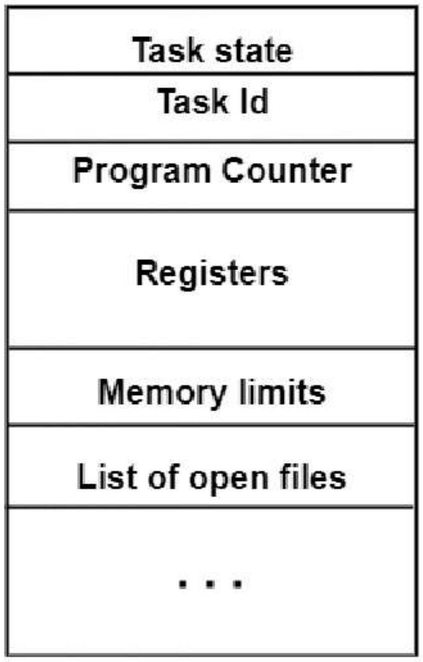


Figure 4.2:Task Control Block

Task Stack: Task stacks reside in RAM and are accessible through stack pointers. They serve as memory areas where tasks can save their execution context during context switching. They store the local variables, function call information, and the task's execution context during context switches.

Task Routine: Task routines contain the actual program code that tasks execute. These routines typically reside in ROM (Read-Only Memory).

Scheduler: The scheduler plays a pivotal role in task management. It keeps track of the state of each task, including whether they are ready to execute, and allocates CPU time to one of the ready tasks. Various scheduling algorithms are employed in RTOS to determine the order in which tasks are executed.

Polled Loop: Some RTOS systems use a polled loop to sequentially check if specific tasks require CPU time. This approach ensures that tasks are executed in a predetermined order based on their readiness.

RTOS task management is a critical aspect of real-time systems. It involves breaking down complex applications into manageable tasks with well-defined time properties, managing their states, and utilizing context switching and scheduling algorithms to ensure that tasks execute correctly and meet deadlines. Task management is fundamental for achieving real-time responsiveness and predictable behavior in embedded systems and critical applications.

4.4 Realtime Scheduling

In soft real-time systems, the scheduling prioritizes critical processes over noncritical ones, but it doesn't provide a guarantee of when these critical processes will be scheduled. In contrast, hard real-time systems have more stringent requirements. In a hard real-time system, a task must be serviced by the CPU precisely within its deadline. Any service provided after the deadline, is as good as no service at all.
Now, let's delve into the specifics of scheduling for hard real-time systems. However, before we explore individual scheduling algorithms, we need to establish certain characteristics of the processes that are subject to scheduling.

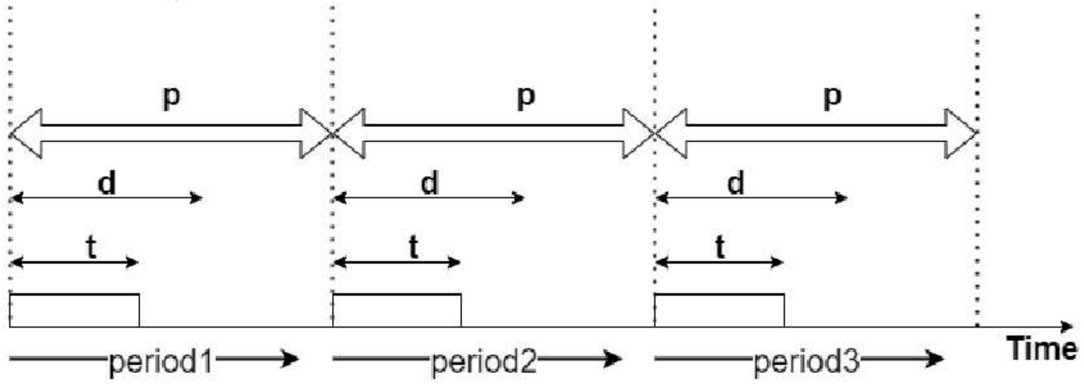


Figure 4.3 : Periodic Task

Processes in hard real-time systems are considered periodic. This means they require CPU execution at fixed intervals, known as "periods." Each periodic process has a predetermined processing time (execution time) denoted as "t" once it acquires the CPU, as shown in figure 4.3.
There's a strict deadline "d" by which time the process must be serviced by the CPU.
Additionally, each periodic process operates within a specified period "p," indicating the time interval between consecutive executions.
The relationship between processing time, deadline, and period is expressed as: $0 <= t <= d <= p$.
The rate of a periodic task is calculated as $1 / p$.
The execution pattern of a periodic process over time adheres to these strict periodic and deadline constraints. Schedulers in hard real-time systems can leverage this relationship to assign priorities based on the deadline or rate requirements of each periodic process. What sets this form of scheduling apart is that a process may need to communicate its deadline requirements to the scheduler explicitly.

Subsequently, the scheduler, utilizing an admission-control algorithm, has two options:

- It can admit the process, guaranteeing that it will complete within its deadline.
- Alternatively, if the scheduler cannot ensure that the task will be serviced within its deadline, it may reject the request as impossible.

4.4.1 Priority-driven scheduling algorithms

Priority-driven scheduling algorithms are commonly used in real-time operating systems (RTOS) to determine the order in which tasks or processes are executed based on their assigned priorities. In these algorithms, each task is associated with a priority value, and the task with the highest priority gets to execute first. Priority-driven scheduling algorithms are particularly useful in systems where tasks have varying levels of importance or deadlines to meet.
Here are some key features and aspects of priority-driven scheduling algorithms:

Priority Levels: Each task is assigned a priority level that reflects its importance or urgency within the system. Lower numerical values often indicate higher priorities, where a lower number means a task is more critical.

Preemption: Priority-driven scheduling algorithms may or may not allow preemption. Preemption means that a higher-priority task can interrupt the execution of a lower-priority task. Non-preemptive priority-driven scheduling completes the execution of the currently running task before allowing a higher-priority task to run, whereas preemptive priority-driven scheduling immediately switches to the higher-priority task when it becomes available.

Fixed vs. Dynamic Priorities: Priorities can be fixed, where they are assigned when tasks are created and do not change during runtime. Alternatively, they can be dynamic, with priorities recalculated dynamically based on task behavior, deadlines, or other factors.

Schedulability Analysis: In real-time systems, schedulability analysis is often performed to ensure that the tasks can meet their deadlines based on their priorities and execution times. Priority-driven algorithms can provide guarantees of meeting deadlines when properly configured.

We will delve into various scheduling algorithms designed explicitly to meet the stringent deadline requirements of hard real-time systems. These algorithms play a crucial role in ensuring that critical tasks are executed within their specified deadlines, making them well-suited for applications where timing guarantees are important.

4.4.1.1 Rate Monotonic Scheduling

The rate-monotonic scheduling algorithm employs a static priority approach with preemption to schedule periodic tasks. When a higher-priority task becomes available while a lower-priority task is running, the higher-priority task will preempt the lower-priority one. Upon entry into the system, each periodic task is assigned a priority inversely proportional to its period. In simpler terms, tasks with shorter periods receive higher priorities, while those with longer periods are assigned lower priorities. This prioritization strategy is based on the idea of giving higher importance to tasks that need CPU access more frequently. Moreover, rate-monotonic scheduling assumes that the processing time of a periodic process remains consistent for each CPU burst. In other words, whenever a process gains CPU control, its CPU burst duration remains the same. This assumption simplifies scheduling decisions and aids in meeting the real-time requirements of the system.

Rate-Monotonic Scheduling (RMS) is a commonly used static priority scheduling algorithm in real-time systems. It aims to schedule periodic tasks in a way that maximizes system efficiency while ensuring that tasks meet their deadlines.

Key Concepts of RMS:

Static Priority Policy with Preemption:

RMS operates on a static priority policy, where each task is assigned a priority based on its periodicity. Preemption is allowed, meaning that a higher-priority task can interrupt the execution of a lower-priority task.

Priority Assignment Based on Periods:

Upon task creation, RMS assigns priorities inversely proportional to the task's period.

Shorter periods result in higher priorities, while longer periods receive lower priorities.

The rationale behind this is to prioritize tasks that require CPU execution more frequently.

Uniform Processing Time:

RMS assumes that the processing time of a periodic process remains constant for each CPU burst.

In other words, each time a process acquires the CPU, it takes the same amount of time to complete its execution.

Illustration with Examples:

Let's consider an example involving two processes, P1 and P2, with the following characteristics:

- P1: Period (p1) = 50, Processing Time (t1) = 20
- P2: Period (p2) = 100, Processing Time (t2) = 35

Scheduling with Priority Assignment (P2 > P1):

If P2 is assigned a higher priority than P1, the execution timeline shows that P2 starts first and completes at time 35. However, P1 misses its deadline at time 50, as shown in figure 4.4 ,even though there is available CPU time.

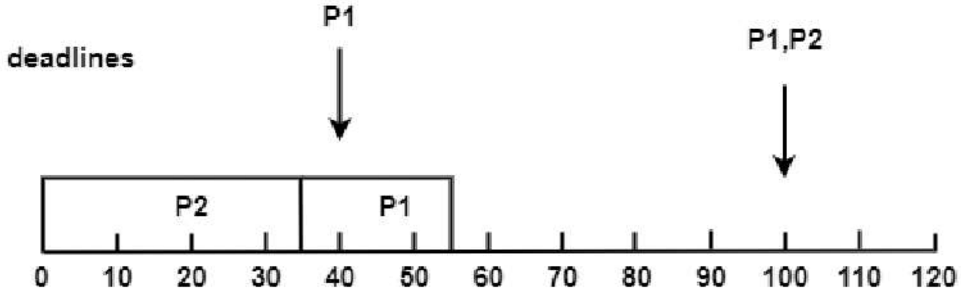


Figure 4.4 : Periodic Task-priority assigned

Scheduling with Rate-Monotonic Scheduling (P1 > P2):

In RMS, P1 is assigned a higher priority than P2 because its period (50) is shorter. With RMS, P1 starts first and completes its CPU burst at time 20, meeting its first deadline. P2 starts running at this point, runs until time 50, and then gets preempted by P1. P1 completes its CPU burst at time 70, and the scheduler resumes P2. P2 completes its CPU burst at time 75, also meeting its first deadline.Figure 4.5 shows this.

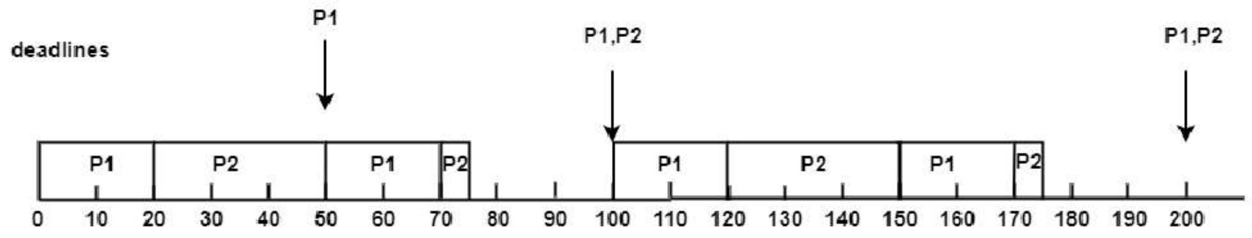


Figure 4.5 : Rate Monotonic Scheduling

Optimality and Limitations:

RMS is considered optimal, meaning that if a set of processes cannot be scheduled by this algorithm, no other algorithm that assigns static priorities can schedule them either. However, RMS has limitations, primarily that CPU utilization is bounded. The worst-case CPU utilization for scheduling N processes is $N(2^{(1/N)} - 1)$. As the number of processes increases, CPU utilization decreases, which means that RMS may not fully maximize CPU resources in all cases. Rate-Monotonic Scheduling assigns task priorities based on their periods, ensuring that tasks with shorter periods receive higher priorities. While it is optimal, it has limits on CPU utilization, making it suitable for systems with bounded resource requirements and stringent timing constraints.

4.4.1.2 Earliest-Deadline-First (EDF) Scheduling:

The Earliest-Deadline-First (EDF) scheduling algorithm is a dynamic priority scheduling scheme used in real-time systems. Unlike static priority algorithms like rate-monotonic scheduling (RMS), EDF dynamically assigns priorities to tasks based on their deadlines. In EDF, tasks with earlier deadlines are given higher priorities, while those with later deadlines receive lower priorities.

Key Concepts of EDF:

Dynamic Priority Assignment:

EDF dynamically adjusts task priorities based on their deadlines. When a task becomes runnable, it must specify its deadline requirements to the system.

Unlike RMS, where priorities remain fixed, EDF is more flexible in adapting priorities to changing deadlines.

Scheduling Illustration:

Consider the example involving two processes, P1 and P2, which couldn't meet their deadlines under rate-monotonic scheduling.
P1: Period (p1) = 50, Processing Time (t1) = 25
P2: Period (p2) = 80, Processing Time (t2) = 35

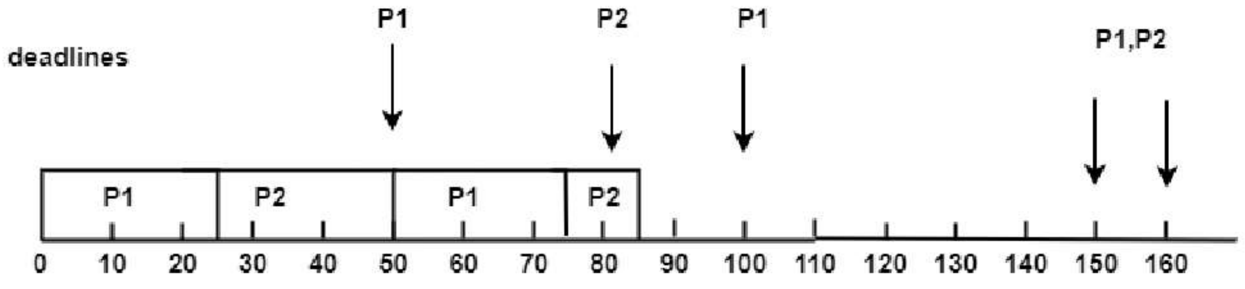


Figure 4.6 : Missed Deadline-RMS

In EDF scheduling, process P1 initially has a higher priority because it has the earliest deadline.
P2 starts running at the end of P1's CPU burst but continues running. Unlike RMS, which would allow P1 to preempt P2 at the start of its next period, EDF lets P2 continue because its next deadline is earlier than that of P1.

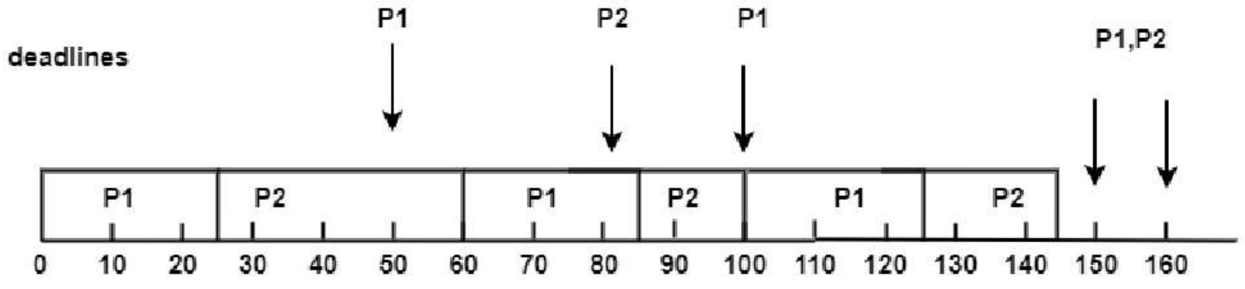


Figure 4.7 : Earliest Deadline First

Priorities change dynamically, ensuring both P1 and P2 meet their deadlines.
The scheduler switches tasks based on which one has the earliest deadline, thus achieving optimal scheduling theoretically.

Flexibility and Non-Periodic Tasks:

EDF does not require tasks to be periodic, nor do they need to have constant CPU burst times. The essential requirement is that a task announces its deadline when it becomes runnable.
This flexibility makes EDF suitable for handling a wide range of real-time task characteristics.

Theoretical Optimality:
EDF is theoretically optimal, meaning that it can schedule processes in a way that allows each process to meet its deadline requirements while achieving 100 percent CPU utilization.

However, in practical implementations, achieving 100 percent CPU utilization is challenging due to the overhead associated with context switching between processes and handling interrupts.
Earliest-Deadline-First (EDF) scheduling dynamically assigns task priorities based on their deadlines, making it highly flexible and theoretically optimal for real-time systems. While it can achieve maximum CPU utilization in theory, practical implementations may fall short due to context-switching and interrupt-handling overhead. EDF is particularly well-suited for systems with diverse task characteristics and changing deadlines.

4.4.1.3 Proportional Share Scheduling:

Proportional share schedulers operate by dividing a total number of shares, denoted as "T," among all running applications or processes. Each application is assigned a certain number of shares, "N," ensuring that it will receive N/T of the total processor time. This approach helps in fairly distributing CPU resources among applications based on their allocated shares.

Example:
Let's illustrate this with an example. Suppose we have a total of T = 100 shares to allocate among three processes, A, B, and C. The allocation is as follows:
Process A is assigned 50 shares.
Process B is assigned 15 shares.
Process C is assigned 20 shares.

This allocation guarantees that:
Process A will receive 50 percent of the total processor time.
Process B will receive 15 percent.
Process C will receive 20 percent.

Admission-Control Policy:
Proportional share schedulers must work in conjunction with an admission-control policy to ensure that an application receives its allocated shares of time. The admission-control policy assesses whether a new client requesting a specific number of shares can be admitted based on the availability of shares.

In the example mentioned earlier, the total allocated shares are 50 (A) + 15 (B) + 20 (C) = 85 shares out of the total of T = 100 shares. Therefore, if a new client requests shares, the admission-control policy would only approve the request if there are sufficient shares available to maintain the proportional allocation for existing processes.
In this way, proportional share scheduling ensures that applications or processes receive CPU time based on their assigned shares, promoting fairness and resource allocation in a predictable manner.
This approach is particularly useful in multi-user systems or environments where different applications or users have varying resource requirements and priorities.

4.5 Example RTOS

Categories of Existing RTOS:

Priority-Based RTOS for Embedded Applications:
The first category of existing RTOS focuses on priority-based kernels tailored for embedded systems. Examples of such RTOS include OSE, VxWorks, QNX, VRTX32, pSOS, among others. Many of these RTOS solutions are commercially available. In this category, applications are designed and programmed with priority-based scheduling in mind. This means tasks are assigned priorities, and deadlines are often used as a basis for these priorities. The RTOS ensures that higher-priority tasks are executed before lower-priority ones to meet critical timing requirements.

Real-Time Extensions of Time-Sharing Operating Systems:
The second category encompasses real-time extensions applied to existing time-sharing operating systems. Examples include Real-Time Linux and Real-Time Windows (such as Real-Time NT). These real-time extensions introduce features like locking real-time tasks in main memory and assigning them the highest priorities. These modifications enable time-sharing operating systems, originally designed for general-purpose computing, to accommodate real-time tasks efficiently.

Research RT Kernels:
The third category comprises research-driven RT Kernels, which are often developed for specialized applications or experimentation. Examples include SHARK and TinyOS. These RT Kernels may explore innovative approaches to real-time system design and may not necessarily be intended for mainstream commercial use. They serve as testbeds for novel real-time concepts and technologies.

Run-Time Systems for Real-Time Programming Languages:
The fourth category encompasses run-time systems that support real-time programming languages. Examples include Ada, Erlang, and Real-Time Java. These languages are specifically designed to facilitate real-time programming, and the run-time systems associated with them ensure that real-time constraints, such as timing and predictability, are met. They are often used in critical systems where reliability and determinism are paramount.

4.5.1 VxWorks

VxWorks is a well-known real-time operating system renowned for its robust support of hard real-time requirements. Developed by Wind River Systems, this commercial RTOS finds extensive utilization across various domains, including the automotive industry, consumer electronics, industrial devices, and networking equipment like switches and routers. Remarkably, VxWorks was also chosen to command the Mars rovers, Spirit and Opportunity, during their exploration missions on the Red Planet in 2004.

The Wind microkernel extends support for individual processes and threads, using the Pthread API for thread management. However, it's noteworthy that, akin to Linux, VxWorks opts not to differentiate between processes and threads; instead, it collectively refers to both as "tasks." This unified approach simplifies task management within the operating system.

VxWorks Features:

- High Performance:
  VxWorks is renowned for its high-performance capabilities, making it well-suited for applications where responsiveness and efficiency are critical.

- Unix-like Multitasking:
  This RTOS offers Unix-like multitasking support, allowing multiple tasks to run concurrently, efficiently sharing system resources.

- Scalable and Hierarchical Environment:
  VxWorks provides a scalable and hierarchical RTOS environment. This means it can be tailored to suit the needs of various applications, from small embedded systems to large and complex ones.

- Host and Target-Based Development:
  VxWorks supports a host and target-based development approach. This approach facilitates the development and testing of software on a host system before deploying it to the target hardware.

- Device Software Optimization:
  VxWorks introduces Device Software Optimization, a methodology that accelerates the development and execution of device software, improving its quality and reliability.

VxWorks RTOS Kernel:

- Processor Abstraction Layer (PAL):
  VxWorks 6.x features a Processor Abstraction Layer (PAL). This layer enables application development that is easily portable to new hardware versions. Developers can design applications for future hardware versions by simply changing the PAL hardware interface.

- Advanced Processor Support:
  VxWorks supports a wide range of advanced processor architectures, including ARM, Cold Fire, MIPS, Intel, and SuperH, ensuring compatibility with various hardware platforms.

- Hard Real-Time Applications:
  VxWorks is well-suited for hard real-time applications, where strict timing requirements must be met consistently.

- Kernel Mode Execution of Tasks:
  It supports kernel mode execution of tasks, allowing for efficient and direct access to system resources while maintaining control over hardware.

- Open Source and Protocol Support:
  VxWorks supports open-source Linux and the TIPC (Transparent Inter-Process Communication) protocol, enhancing its versatility and connectivity options.

- Preemption Points at Kernel:
  VxWorks provides preemption points within the kernel, enabling the system to respond quickly to higher-priority tasks or interrupts.

- Scheduling Options:
  VxWorks offers both preemptive and round-robin scheduling, allowing developers to choose the scheduling mechanism that best fits their application's needs.

- POSIX and UNIX Standards:
  It supports POSIX (Portable Operating System Interface) standard asynchronous I/O operations and UNIX standard buffered I/O operations, enhancing interoperability with existing software.

- PTTS 1.1 Compliance:
  VxWorks complies with PTTS 1.1 (Portable Test and Target Software) standards since December 2007, ensuring adherence to industry standards.

- Inter-Process Communication (IPC):
  VxWorks offers various IPC mechanisms, including TIPC for network and clustered system environments, as well as POSIX 1003.1b standard IPCs and interfaces, promoting efficient communication between tasks and processes.

- Separate Context for Tasks and ISRs:
  VxWorks maintains separate execution contexts for tasks and Interrupt Service Routines (ISRs), ensuring efficient and predictable handling of hardware interrupts and software tasks.

In essence, VxWorks is a versatile and high-performance RTOS that excels in supporting a wide range of applications, from embedded systems to complex real-time environments. Its features encompass scalability, hardware abstraction, support for advanced processors, and adherence to industry standards, making it a valuable choice for developers in various domains.

4.5.2 RT Linux

RT-Linux is an operating system that exemplifies the concept of a real-time operating system (RTOS). In RT-Linux, a compact real-time kernel coexists harmoniously with a standard Linux kernel, resulting in a hybrid operating environment that caters to both real-time and general-purpose computing needs.

Key Characteristics of RT-Linux:

- Integration of Real-Time and Standard Linux Kernel:
  In RT-Linux, the real-time kernel is strategically positioned between the standard Linux kernel and the hardware. From the perspective of the standard Linux kernel, this real-time layer appears as actual hardware. This arrangement allows for the seamless operation of real-time tasks alongside standard Linux processes.

- Interrupt Handling:
  The real-time kernel takes control of all hardware interrupts. Specifically, it intercepts interrupts related to real-time tasks and ensures that the appropriate Interrupt Service Routines (ISRs) are executed promptly. For non-real-time tasks, these interrupts are queued and presented to the standard Linux kernel as software interrupts during its execution.

- Priority Assignment:
  The real-time kernel assigns the lowest priority to the standard Linux kernel. Consequently, real-time tasks take precedence in execution, ensuring that they meet their stringent timing requirements.

- User-Controlled Real-Time Tasks:
  Users have the flexibility to create and manage real-time tasks, allowing them to define scheduling algorithms, task priorities, execution frequencies, and other parameters. This level of control empowers users to achieve precise timing for their real-time applications.

- Privileged Real-Time Tasks:
  Real-time tasks are granted privileged status, affording them direct access to hardware resources. Notably, these tasks do not utilize virtual memory, which can introduce unpredictable latencies.

Contrasting Linux and RT-Linux:

Linux (Non-Real-Time Features):

Linux's scheduling algorithms are not inherently designed for real-time tasks. While they deliver commendable average performance and throughput, they may not guarantee strict real-time deadlines. Linux may exhibit unpredictable delays due to factors like uninterruptible system calls, the use of interrupt disabling, and the support for virtual memory. Context switches in the Linux kernel can consume hundreds of microseconds. The timer resolution in Linux tends to be coarse, with a typical resolution of 10 milliseconds. The Linux kernel is non-preemptible, meaning that once a process or task starts executing, it cannot be preempted until it voluntarily yields control.

RT-Linux (Real-Time Features):
RT-Linux is tailored to support real-time scheduling, ensuring the fulfillment of hard deadlines for critical tasks. Predictable and minimized delay is a hallmark of RT-Linux, primarily due to its small kernel size and restricted operations. RT-Linux offers finer time resolution, allowing for more precise timing control. The real-time kernel in RT-Linux is preemptible, enabling the swift switching between tasks when necessary to meet real-time requirements. Unlike the standard Linux kernel, RT-Linux does not support virtual memory, eliminating potential sources of latency.

In summary, RT-Linux showcases the convergence of real-time and general-purpose computing within a single operating environment. Its real-time features, including precise scheduling, low latency, and preemptibility, make it an ideal choice for applications that demand strict timing and responsiveness, such as industrial automation and embedded systems.

4.6 Self-Assessment Questions

Q1. Explain the evolution of RTOS architecture based on the complexity of applications it supports. [4 marks, L3]

Q2. What are the primary functions of a monolithic kernel, and how does it differ from a microkernel in terms of architecture and performance? [4 marks, L2]

Q3. List some Unix-like operating systems that typically use monolithic kernels. What advantages do monolithic kernels offer? [3 marks, L2]

Q4. In what scenarios might a developer prefer a microkernel over a monolithic kernel or vice versa? Provide real-world examples to illustrate your points. [5 marks, L3]

Q5. Why is meeting deadlines crucial in real-time systems, and how is the "Deadline" property related to task execution? [4 marks, L2]

Q6. Describe the different states a task can exist in within an RTOS, and provide examples of scenarios where each state might be applicable. [5 marks, L2]

Q7. List and briefly explain the key components of a task in an RTOS, including Task Control Block (TCB), Task Stack, Task Routine, and Scheduler. [6 marks, L2]

Q8. Explain the role of the Scheduler in task management, and mention the various scheduling algorithms used in RTOS. [4 marks, L2]

Q9. What is the primary principle behind priority-driven scheduling algorithms, and why are they essential in real-time systems? [6 marks, L2]

Q10. Explain the priority assignment strategy used in Rate Monotonic Scheduling (RMS) and the rationale behind it. [6 marks, L2]

Q11. What is the theoretical optimality of EDF, and why is it considered optimal? [2 marks, L2]

Q12. In which types of environments or systems is proportional share scheduling beneficial? [2 marks, L2]

4.7 Self-Assessment Activities

A1. In a hard real-time system, you have three periodic tasks: Task A with a period of 20 ms and execution time of 8 ms, Task B with a period of 40 ms and execution time of 12 ms, and Task C with a period of 50 ms and execution time of 15 ms. Explain how RMS would prioritize these tasks and whether they can all meet their deadlines.

A2. In an EDF-based scheduling system, there are three tasks: Task P with a deadline of 20 ms, Task Q with a deadline of 15 ms, and Task R with a deadline of 30 ms. Explain how EDF dynamically assigns priorities to these tasks as they become runnable and which task will execute first.

A3. In a shared computing environment with a total of 100 shares, two processes, Process X and Process Y, request CPU time. Process X requests 40 shares, while Process Y requests 30 shares. Can both processes be admitted to run concurrently, and if so, how will the CPU time be allocated between them?

A4. Consider a real-time environment with five tasks, Task A through Task E. Task A has a deadline of 25 ms, Task B has a deadline of 10 ms, Task C has a deadline of 30 ms, Task D has a deadline of 15 ms, and Task E has a deadline of 20 ms. Describe how EDF scheduling would prioritize and execute these tasks as they become available.

A5. In a multi-user system using proportional share scheduling, there are three users: User A with 40 shares, User B with 25 shares, and User C with 35 shares. Explain how the CPU time will be distributed among these users if they all have runnable processes.

4.8 Multiple-Choice Questions

Q1. What is the primary role of an RTOS kernel in a real-time operating system? [1 mark, L1]

A. Managing networking protocols
B. Handling debugging facilities
C. Acting as an abstraction layer between hardware and applications
D. Performing user interface tasks

Q2. Which category of RTOS kernel combines the simplicity of microkernels with some elements of monolithic kernels? [1 mark, L1]

A. Monolithic Kernel
B. Microkernel
C. Hybrid Kernel
D. Exokernel

Q3. Which of the following operating systems typically uses a monolithic kernel? [1 mark, L1]

A. Windows NT
B. QNX
C. Linux
D. BeOS

Q4. In rate-monotonic scheduling, how are task priorities assigned? [1 mark, L1]

A. Based on the task's execution time
B. Based on the task's deadline
C. Based on the task's period
D. Based on the task's arrival time

Q5. Which scheduling algorithm assigns priorities based on a task's deadline, with earlier deadlines receiving higher priorities? [1 mark, L1]

A. Rate-Monotonic Scheduling
B. Round-Robin Scheduling.
C. Earliest-Deadline-First Scheduling.
D. First-Come-First-Serve Scheduling

Q6. In a proportional share scheduler, what does "N/T" represent in terms of CPU time allocation? [1 mark, L1]

A. The number of shares allocated to an application.
B. The total number of shares available
C. The percentage of total processor time allocated to an application.
D. The processing time of a single application.

Q7. Which of the following scheduling algorithms is theoretically optimal for meeting task deadlines and achieving 100 percent CPU utilization? [1 mark, L1]

A. Rate-Monotonic Scheduling.
B. Round-Robin Scheduling.
C. Earliest-Deadline-First Scheduling.
D. First-Come-First-Serve Scheduling

Q8. What is the primary advantage of using proportional share scheduling in RTOS?? [1 mark, L1]

A. Guaranteed hard real-time deadlines.
B. Efficient use of CPU resources.
C. High predictability of context switches
D. Simplified task management

Q9. Why is it important for an RTOS to adapt its architecture as the complexity of applications increases?[1 mark, L1]

A. To reduce the size of the kernel.
B. To minimize hardware interactions
C. To meet the specific requirements of diverse applications.
D. To improve overall system performance.

Q10. In proportional share scheduling, what does "N/T" represent in terms of CPU time allocation? [1 mark, L1]

A. The number of shares allocated to an application.
B. The total number of shares available.
C. The percentage of total processor time allocated to an application.
D. The processing time of a single application

Q11. What is the role of an admission-control policy in proportional share scheduling? [1 mark, L1]

A. It dynamically adjusts task priorities.
B. It divides CPU time among tasks..
C. It guarantees an application receives its allocated shares.
D. It assigns priorities based on deadlines.

Q12. What is the primary function of a scheduler in RTOS task management? [1 mark, L1]

A. Storing task execution context
B. Managing task states
C. Allocating CPU time to tasks
D. Controlling access to RAM

Q13. Which component of an RTOS task typically contains information about the task's state, priority, and execution context?

A. Task Routine
B. Task Stack
C. Task Control Block (TCB)
D. Scheduler

Q14. What is the primary purpose of a Task Control Block (TCB) in an RTOS?

A. Storing task routines
B. Managing local variables
C. Remembering task context
D. Allocating CPU time

4.9 Keys to Multiple-Choice Questions

Q1. Acting as an abstraction layer between hardware and applications (C)
Q2. Hybrid Kernel (C)
Q3. Linux (C)
Q4. Based on the task's period (C)
Q5. Earliest-Deadline-First Scheduling. (C)
Q6. The percentage of total processor time allocated to an application. (C)
Q7. Earliest-Deadline-First Scheduling. (C)
Q8. Efficient use of CPU resources. (B)
Q9. To meet the specific requirements of diverse applications. (C)
Q10. The percentage of total processor time allocated to an application (C)
Q11. It guarantees an application receives its allocated shares. (C)
Q12. Allocating CPU time to tasks. (C)
Q13. Task Control Block (TCB)(C)
Q14. Remembering task context. (C)

4.10 Summary of the Unit

In the domain of Real-Time Operating Systems (RTOS), the architecture adapts according to the complexity of the supported applications. Initially, for simpler applications, an RTOS primarily consists of a kernel. However, as applications become more complex, additional modules like networking protocol stacks, debugging tools, and device I/O functionalities are incorporated alongside the kernel. This evolving RTOS architecture comprises various components to cater to diverse application demands. At the core of an RTOS architecture lies the kernel, serving as a crucial intermediary layer between the underlying hardware and the running applications. The RTOS kernel abstracts complex hardware interactions, shielding application developers from hardware intricacies. This separation allows

developers to focus on application logic without delving into hardware details.
In RTOS, the term "task" is used interchangeably with "thread" or "lightweight process." Tasks in RTOS are sequential programs with timing constraints. Task management is vital in real-time systems for breaking down complex applications into manageable, time-constrained tasks. It involves context switching and scheduling algorithms to ensure tasks meet deadlines, achieving real-time responsiveness in embedded systems and critical applications.
Soft real-time systems prioritize critical processes over noncritical ones but don't guarantee scheduling times.Hard real-time systems require tasks to be serviced by their deadline; missing the deadline is not acceptable.In the context of hard real-time systems, scheduling algorithms ensure tasks meet their deadlines, starting with defining task characteristics. Real-time scheduling algorithms are vital for managing tasks in RTOS environments. Rate-monotonic and earliest-deadline-first scheduling focus on task priorities based on CPU utilization and deadlines, respectively, while proportional share scheduling ensures fairness by dividing CPU time into shares for each task. The choice of scheduling algorithm depends on the specific requirements and characteristics of the real-time system being designed.

4.11 Keywords

Real-time operating systems (RTOS), Priorities,Context switching, Task Control Block (TCB), Real-time scheduling algorithms, Rate-Monotonic Scheduling, Earliest-Deadline-First (EDF) Scheduling, Proportional Share Scheduling

4.12 Recommended Learning Resources

[1] Silberschatz, A., Galvin, P., & Gagne, G. (2005). Operating System Concepts, 7th ed., Hoboken.
[2] William stalling. Operating Systems: Internal and design principles, 7th edition PHI
[3] D.M. Dhamdhere. Operating Systems: A concept-based Approach, 2nd Edition, TMH
[4] https://en.wikipedia.org/wiki/Hybrid_kernel#/media/File:OS-structure2.svg