**Real-Time Operating Systems (RTOS)**

* **Operating System (OS)-**

An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Example of operating system are Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

* Windows is Convenience and gives very good throughput
* Batch OS
* Multi programmed OS
* Multitask or multi sharing OS
* Realtime OS
* Distributed in Environment OS
* Clustered OS
* Embedded OS
* **Priority, Time quantum, I/O request**

**Important functions of an operating System-**

1. **Memory management:** The OS allocates and deallocates memory to different processes, ensuring that each process has sufficient memory to execute without interfering with other processes.
2. **Process management:** The OS manages the execution of multiple processes or threads, scheduling them for execution on the CPU, and ensuring that each process gets a fair share of the CPU's resources.
3. **File management:** The OS provides a file system that allows users and programs to create, access, and manipulate files and directories on storage devices such as hard drives, flash drives, and network drives.
4. **Device management:** The OS provides device drivers and services to manage input/output devices such as keyboards, mice, displays, printers, and network adapters.
5. **Security and protection:** The OS provide security mechanisms to protect the system and user data from unauthorized access and malicious software, such as firewalls, antivirus software, and user account management.
6. **Networking:** The OS provides networking services to enable communication between different devices on a network, such as protocols for sending and receiving data, managing connections, and resolving network addresses.
7. **User interface:** The OS provides a graphical or command-line interface that allows users to interact with the system, launch applications, and perform various tasks.

**Process vs Thread vs Task**

In computer science, a process, a thread, and a task are **all related to the execution of code**, but they are distinct concepts with **different properties and characteristics**.

**A process is an instance of a program that is executing on a computer**. It consists of a set of instructions, data, and resources such as memory, files, and network connections. A process is typically created by the operating system, which provides it with **a dedicated address space, a process ID, and other resources**. Each process runs in **isolation from other processes** and communicates with them through inter-process communication (IPC) mechanisms.

**A thread is a lightweight unit of execution within a process that shares the same memory space as the process.** A thread consists of a sequence of instructions and a set of registers that represent the current state of execution. Threads are typically **used to perform parallel or concurrent tasks within a process**, such as handling user input, updating the display, or performing background computations. **Multiple threads can run within a single process, sharing the same memory and resources.**

**A task is a high-level abstraction that represents a unit of work to be performed**. A task can be implemented as a process or a thread, or as a combination of both, depending on the specific requirements of the application. Tasks can be created and scheduled by an operating system, a middleware layer, or an application framework.

In summary, a process is an instance of a program that runs in its own isolated memory space, a thread is a lightweight unit of execution within a process that shares the same memory space, and a task is a high-level abstraction that represents a unit of work to be performed.

**Note** - pre-emptive scheduling allows the operating system to interrupt a running task and switch to a higher-priority task, while non-pre-emptive scheduling requires a running task to voluntarily yield the CPU to another task.

* **RTOS Fundamentals:**

**Immediate, quick, deadline.**

When we hear a word “operating system (OS)” the first one come our mind are Windows, Linux, MacOS etc for computer & Android, iOS for mobile and many more. RTOS is special type of operating system specifically design for embedded system.

**“A system is said to be Real Time if it is providing a required level of service in a bounded response time”.**

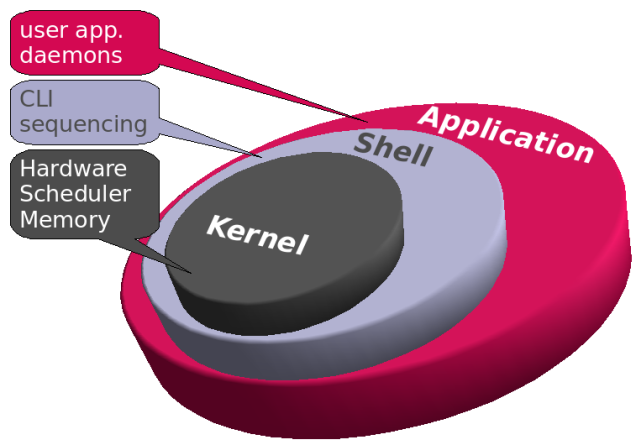
Some examples of RTOSs are VxWorks, QNX, Win CE, pSOS, Nucleus® RTOS, and FreeRTOS, the Zephyr Project.

**Type:**

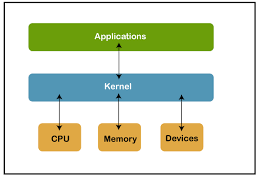
***Hard Real Time*** – Task timings and deadlines are handled very strictly. (eg. critical healthcare, air traffic control, vehicle subsystem control ).

***Soft Real Time* –**it provides some relaxation in the time limit. (eg. online transactions, web site and service)

* **Kernel and shell:**



The heart of every operating system is called as ‘kernel’. Tasks are relieved of monitoring the hardware. It’s the responsibility of the kernel to manage and allocate the resources.



# Shell:

Shell is the interface between user and kernel. It is a command-line interpreter and is the interface between the user and the kernel. The user can enter commands to the shell. Then it interprets the commands to perform the required task. Furthermore, it executes programs and shell scripts. A shell script is a set of commands. The user should follow the standard syntax to write commands to the shell.

* **Difference between OS and RTOS**
* A real-time operating system (RTOS) specializes in extremely quick reaction times, whereas a traditional operating system (OS) concentrates on sequentially computing throughout the whole array of processes.
* **Hardware and Economical factors**: An RTOS is usually designed for a low end, standalone device like an ATM, Vending machines, Kiosks etc. RTOS is light weight and small in size compared to a GPOS. A GPOS is made for high end, general purpose systems like a personal computer, a work station, a server system etc. The basic difference between a low-end system and high end system is in it’s hardware configuration.
* **Task scheduling: In** the case of a GPOS – task scheduling is not based on  “priority” always! GPOS is programmed to handle scheduling in such a way that it manages to achieve high throughput.  Where as in an RTOS – scheduling is always priority based. Most RTOS uses pre-emptive task scheduling method which is based on priority levels. Here a high priority process gets executed over the low priority ones.
* **Latency**: GPOS is unbounded dispatch latency, which most GPOS falls into. The more number of threads to schedule, latencies will get added up! An RTOS has no such issues because all the process and threads in it has got bounded latencies – which means – a process/thread will get executed within a specified time limit.
* **Preemptible** **Kernel**: The kernel of an RTOS is preemptible where as a GPOS kernel is not preemptible. This is a major issue when it comes to serving high priority process/threads first. If kernel is not preemptible, then a request/call from kernel will dominate all other process and threads. For example:- a request from a driver or some other system service comes in, it is treated as a kernel call which will be served immediately overriding all other process and threads. In an RTOS the kernel is kept very simple and only very important service requests are kept within the kernel call.

* **When do you need an RTOS?**
* An RTOS is necessary when there are several processes and devices, and the **processes’ timing is more important than average performance**. **If you need multiple processes to run on a schedule, you need an RTOS.**
* An RTOS can **effectively handle interrupts based on priority to control scheduling.**
* Some of the characteristics of an RTOS are that it must be multithreaded, pre-emptive, and provide for thread priority. An RTOS must also include a system of priority inheritance, predictably sustain thread synchronization, and have a mechanism to avoid [priority inversion](https://en.wikipedia.org/wiki/Priority_inversion).
* **Characteristics of RTOS-**
  + Consistency
  + Reliability
  + Scalability
  + Performance
  + Predictability
* **FUNCTION OF RTOS-**
  + Task management
  + Scheduling resource allocation
  + Interrupt handling

1. **Task management –**

* In real time application, any process which takes a space and execution time and occupies predefined amount of memory is called task.
* Task management is the process of managing tasks through its life cycle.
* Task operation-
* controlling task scheduling
* Creating and deleting tasks

1. **Task state**

* **Suspended State**-

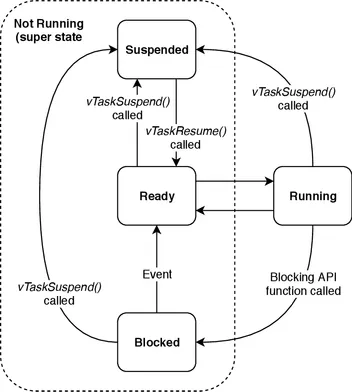
It is one of the state of not running task. We can suspend any task by using the function**vTaskSuspend()**and to use the same task again we need to can resume the task by using the function**vTaskResume()**.

* **Blocked State-**

Any task can be in blocked state due to

* + if the task is delayed
  + interrupt waiting
  + resources waiting
* **Ready state-**

For any task to be in running state it must be in ready state first where scheduling is done. All the task in the ready state are ready for execution as soon as processor picks them as per the scheduling policy.



### Free RTOS task creation

In freeRTOS we create task by using **xCreateTask** . It takes 5 arguments which defines various features of the task.

xTaskCreate(MyTask\_pointer, "task\_name", stack depth, Parameter, Priority, TaskHandle);

#### MyTask\_pointer: It is pointer to task entry function. we need to define the task with certain function so, it is just the name of the function which is used while creating a task. task\_name

A descriptive name for the task. This is mainly used to facilitate debugging.

**stack depth:** *The size of the task stack specified as the number of variables the stack can hold – not the number of bytes. we should make sure to select stack size according to the complexity of computation. For example, if the stack is 16 bits wide and us Stack Depth is defined as 100, 200 bytes will be allocated for stack storage.*

**Parameter:** *Pointer that will be used as the parameter for the task being created .If we want to pass a pointer to a variable as an argument to the task function, we can use this argument. Otherwise, we can pass the NULL value.*

**Priority:** *If we create any task than its priority is very important to be set. The priority of any task is set by user according to its complexity and the requirement of the project.For**example, if we create four tasks and assign them priority 0, 1,2 and 3. Hence, zero means the lowest priority and 3 means the highest priority.*

#### TaskHandle: It is used to pass a handle to the created task. This argument keeps the handle of the function that we can use to change function features such as the deletion of a task, changing its priority, etc.

## Important Functions In FREE RTOS

**xTaskCreate()**– for task creation

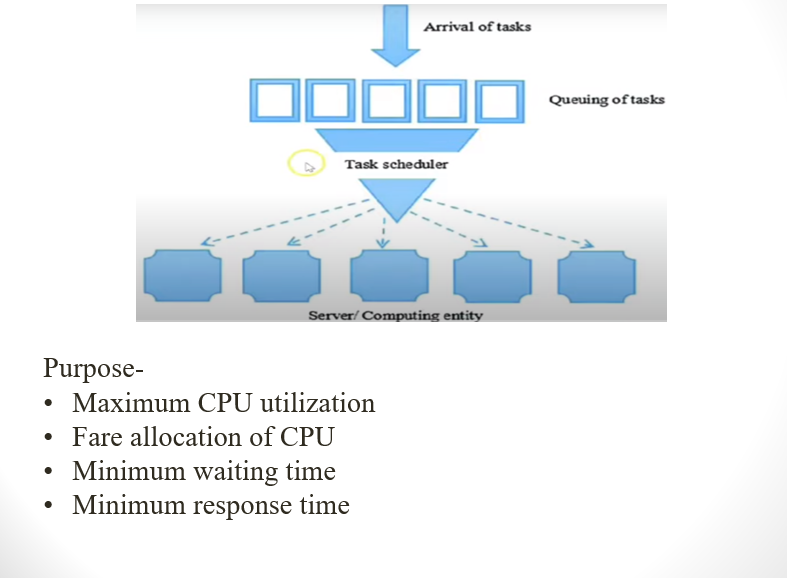
**vTaskSuspend()**– for task suspension

**vTaskResume()**– for resuming the task

**vTaskDelay()**– for delaying a task

**vTaskDelete()**– for deleting a task

1. **Task scheduling-**



Way: a process needs CPU time and I/O time both for its execution. In a **multi-programming system**, one process can use CPU while another process is waiting for I/O whereas, on the other hand in **a uni programming system**, all the time get wasted in waiting for I/O whereas CPU is free during that time.

### Types of Scheduling policies:

**Pre-emptive Scheduling:** In this type of scheduling, tasks run with equal time slice without considering the priorities.

**Priority-based Pre-emptive:** High priority task will run first.

**Co-operative Scheduling:** Context switching will happen only with the co-operation of running tasks. Task will run continuously until task yield is called.

* **Scheduling algorithm**

1. **First come first serve(FCFS):**

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive, scheduling algorithm.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.

Table

Description automatically generatedDiagram, table

Description automatically generated with medium confidence

1. **Shortest job first (SJF)-**

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processer should know in advance how much time process will take.

**Table

Description automatically generated**

1. **Priority scheduling-**

* Priority scheduling is a non-pre-emptive algorithm and one of the most common scheduling algorithms in batch systems.
* Each process is assigned a priority. Process with highest priority is to be executed first and so on.
* Processes with same priority are executed on first come first served basis.
* Priority can be decided based on memory requirements, time requirements or any other resource requirement

Table

Description automatically generated

1. **Round robin scheduling-**

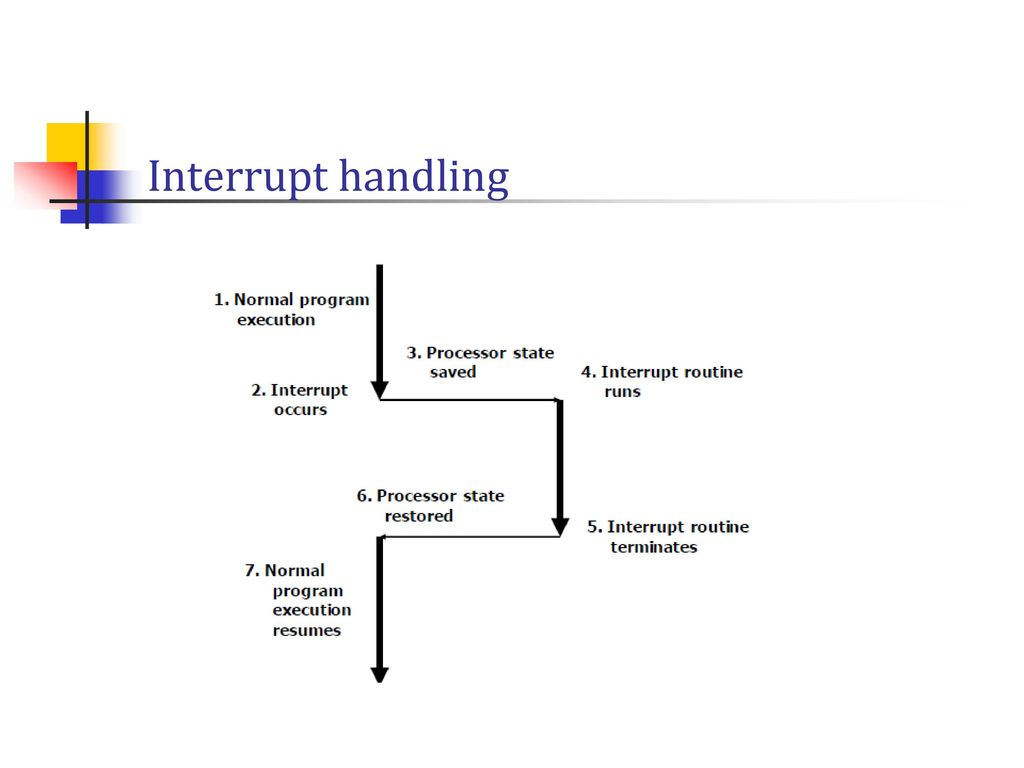
* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a **quantum**.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.

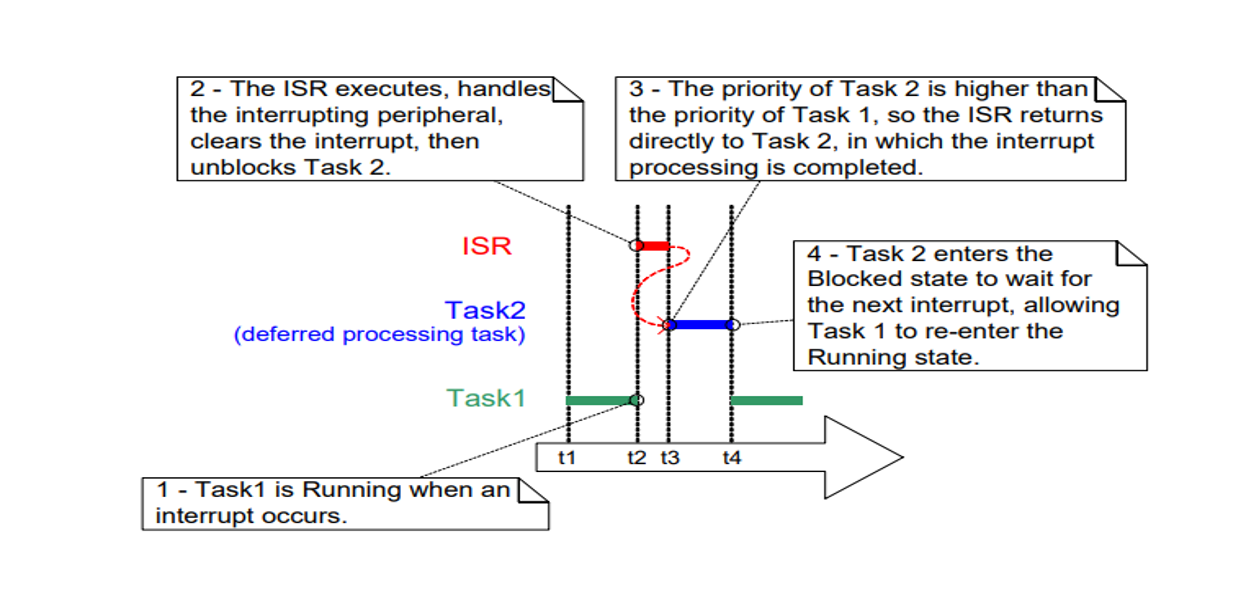
Table

Description automatically generated

1. **Interrupt handling-**

An interrupt is a signal from a device attached to a computer or from a program with a computer that causes the main program that is operating system to stop and figure out what to do next.





1. **Task synchronization:**

**Semaphore:** **Semaphore** in OS is an integer value that indicates whether the resource required by the process is available or not. The value of a semaphore is modified by wait() or signal() operation.

* **Wait**

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Wait(Semaphore S)

{

S.vlaue= S.value -1;

If (S.value≤0)

{

block the process Sleep();

}

**else**

return;

}

* **Signal**

The signal operation increments the value of its argument S.

Signal(Semaphore S)

{

S.value= S.value + 1;

If(S.value≤0)

{

Unblock the process wake up();

}

}

**Type of semaphore:**

1. **Counting Semaphore**  
The counting semaphores controls access to resources that have finite instances. Therefore the value of counting semaphore is not restricted to a certain domain.

The counting semaphore is initialized with a value equivalent to the number of resources available. When a process wants to access the resource it executes the entry section code where the wait() operation decrements the value of semaphore and allot the resource to the process.

When the counting semaphore value decreases to 0 it means no resources are available now and the process that executes the entry section code further would be blocked. Now when a process releases a resource it executes its exit section code where the signal() operation would increment the value of counting semaphore and release a blocked process which can further try to execute the entry section code.

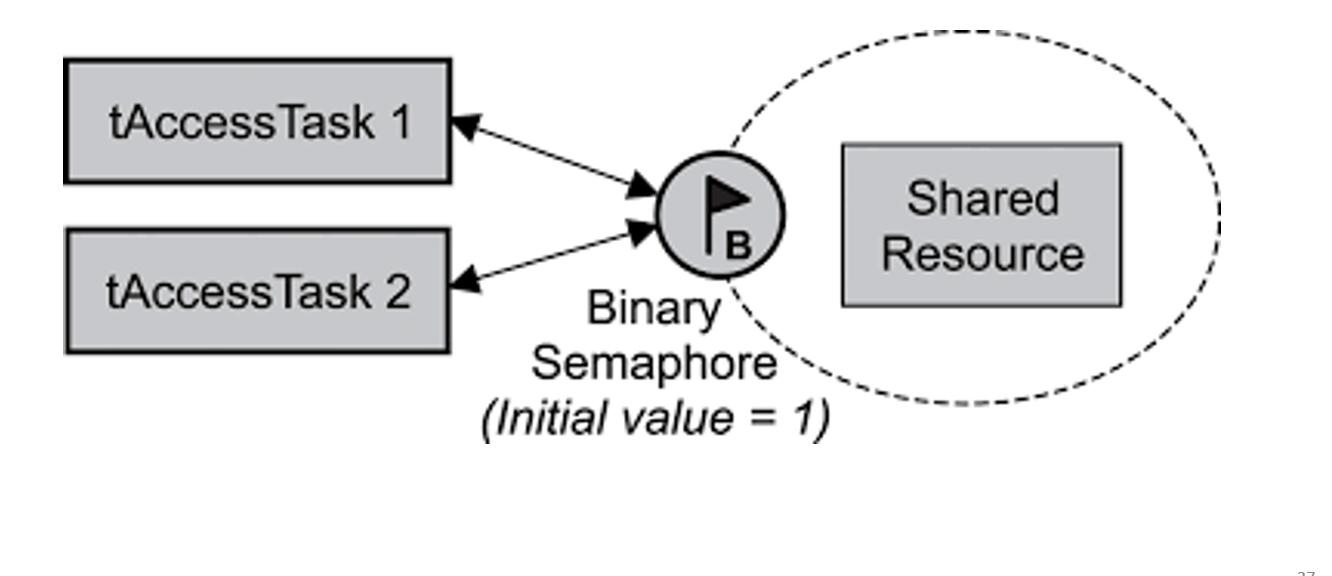
Diagram

Description automatically generated

2. **Binary Semaphores**  
Binary semaphores are generally used to access the critical section. As we know that only one process can enter a critical section at a time therefore the value of binary semaphores ranges over 0 to 1.

Consider that the binary semaphore value is initialized to 1. Now if a process P1 want to enter the critical section then it executes the entry section code where the wait() operation decrements the semaphore value to 0 and enters the critical section. Now when second process P2 tries to enter the critical section then it executes the entry section code where the wait() operation verifies the binary semaphore value is already 0 which means no process further can enter the critical section and blocks the second process P2.

When process P1 leaves the critical section it executes the signal() operation of exit section which increments the semaphore value to 1 and releases a blocked process P2 which can further try to execute entry section code to enter the critical section.



**Mutex:**

**Mutex**is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

Diagram

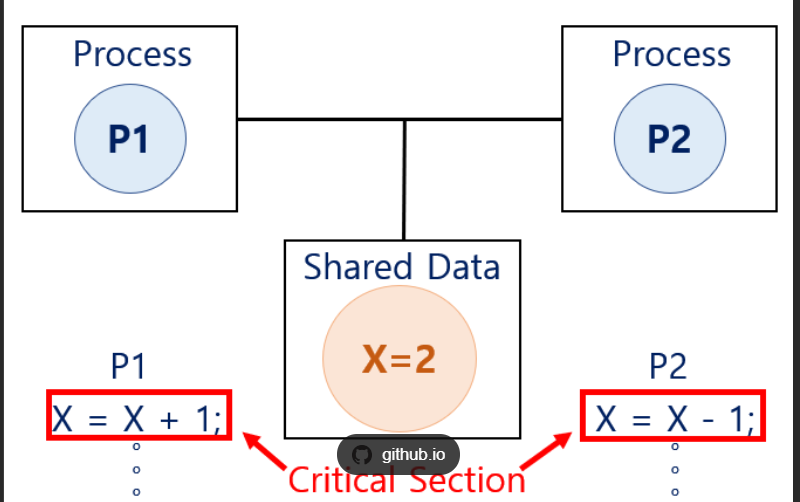
Description automatically generated

**Note:**

**Process and thread:**   
A process is a program under execution i.e an active program. Or A process is the execution of a program that allows you to perform the appropriate actions specified in a program. It can be defined as an execution unit where a program runs. The OS helps you to create, schedule, and terminates the processes which is used by CPU. The other processes created by the main process are called child process

Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming. A thread is lightweight and can be managed independently by a scheduler. It helps you to improve the application performance using parallelism.

**Critical section-** the segment of code where processes access shared resources, such as common variables and files, and perform write operations on them.



**Race condition**: A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.

## Disadvantages of Real Time Operating System

• **Limited tasks**: RTOS can run very limited tasks simultaneously and it concentrates on only those applications which contain error or interrupts so that it can avoid them. Due to this, some tasks have to wait for unlimited time.  
• **Low multi-tasking**: As RTOS is the system which concentrates on few tasks. So sometimes it is difficult for these systems to do multi-tasking.  
• **Low priority of tasks**: The tasks which has the low priority must wait for a long time as the RTOS maintain the accuracy of the current programs which are under execution.

* **Cost**: RTOS is often more expensive than general-purpose operating systems, as it requires specialized hardware and software to provide the necessary real-time performance and determinism. - **Expensive**
* **Complexity:** RTOS can be more complex to develop, configure, and maintain than general-purpose operating systems, as it requires expertise in real-time programming, scheduling algorithms, and hardware interfacing.
  + **Not easy to program**
  + **Complex algorithms**: it makes use of some complex algorithms so that the system can give the desired output. These complex algorithms are difficult to understand.
* **Limited hardware support:** RTOS may not be compatible with all hardware platforms, limiting the choice of hardware and making it difficult to switch between platforms.
* **Limited software ecosystem:** The software ecosystem for RTOS is often smaller than that of general-purpose operating systems, making it harder to find pre-existing software libraries and tools.
* **Overhead:** RTOS can introduce overhead in terms of processing time and memory usage, which may be problematic for applications with strict resource constraints.

**Queue**

* A queue is a data structure that allows one task to send data to another task in a thread-safe manner. In FreeRTOS, a task can send data to a queue using the **xQueueSend()** function, and another task can receive the data using the **xQueueReceive()** function. The queue ensures that the data is transmitted safely and efficiently between tasks, even in a multitasking environment.
* Queues can be used for synchronization between tasks by allowing one task to block until another task sends data to the queue. For example, a producer task may send data to a queue, while a consumer task waits for data to become available in the queue using the xQueueReceive() function. The consumer task will block until data is available, which effectively synchronizes the tasks.
* **FreeRTOS also provides other synchronization mechanisms, such as semaphores and mutexes**, which can be used for similar purposes. The choice of synchronization mechanism depends on the specific requirements of the application and the type of data being transmitted between tasks.

**BSP (Board Support Package)**

* **A Board Support Package (BSP) is a collection of software components and drivers that enable a particular hardware platform or board to be used with an operating system.** BSPs are typically used in embedded systems, such as those used in industrial control, automotive, or consumer electronics applications.
* **A BSP includes low-level device drivers for hardware components, such as the processor, memory, input/output interfaces, and peripherals like sensors or displays**. It also includes software components that provide the necessary functionality for the hardware to be used with an operating system, such as bootloaders, interrupt handlers, and initialization routines.
* **BSPs are typically provided by the manufacturer of the hardware platform or board, or by third-party vendors**. The BSP can be customized for specific applications, by adding or removing components, optimizing driver code for performance, or modifying the kernel configuration.
* **The BSP is an essential component in the development of an embedded system**, as it provides a common software platform that can be used by application developers and system integrators. By providing a pre-configured software environment for a specific hardware platform, **BSPs help to reduce development time and costs**, and ensure that the system is stable and reliable.
* BSPs are often available for a range of operating systems, such as Linux, Windows Embedded, or real-time operating systems like FreeRTOS or VxWorks.

**Trace Hook & Idle Hook**

* Trace hook and idle hook are two types of functions used in computer programming and software development.
* **Trace hook functions are used for debugging and profiling purposes.** They are called whenever the program execution enters or leaves a function or a line of code. Trace hooks can be used to collect data on program execution, such as the number of times a function is called, or the time spent executing a certain line of code. This information can then be used to optimize the performance of the program.
* Idle hook functions are used to perform background tasks or maintenance tasks when the program is idle or not actively processing data. They are typically called in a loop whenever the program is idle, waiting for user input or performing other non-intensive tasks. **Idle hooks can be used to perform tasks such as garbage collection, updating the user interface, or checking for updates.**
* So, trace hook functions are used for debugging and profiling, while idle hook functions are used for performing background tasks during periods of idle time.

**Profiling**

* Profiling is a technique used in computer programming and software development to **analyse the performance of a program**. Profiling involves measuring various metrics of a program, such as the execution time of individual functions, the amount of memory used by the program, and the frequency of function calls.
* The purpose of profiling is to identify performance bottlenecks and **optimize the program's performance**. **By analysing the data collected during profiling**, developers can **identify which parts of the program are consuming the most resources and causing delays or slowdowns**. This information can be used to optimize the program's algorithms, improve memory usage, and reduce the time spent on resource-intensive tasks.
* There are several tools available for profiling, including built-in profiling features in programming languages and third-party profiling software. **Profiling can be done during development, testing, and production stages to ensure that the program is running efficiently and meets the required performance standards.**

**STM32CubeIDE with Profiling**

STM 32CubeIDE is an integrated development environment (IDE) that supports profiling for embedded systems based on STM32 microcontrollers. The following steps can be followed to perform profiling in ST32CubeIDE:

1. Enable profiling: To enable profiling, open the project properties in ST32CubeIDE, go to "C/C++ Build" and then "Settings". In the "Tool Settings" tab, select "Profiling" and check the "Enable profiling" box.
2. Configure profiling options: In the same "Profiling" settings window, you can configure various options for profiling, such as the profiling type (e.g. function profiling, instruction profiling), the profiling interval, and the output format.
3. Build and run the project: Build and run the project in debug mode. Profiling data will be collected during the program execution.
4. View profiling results: After the program execution is complete, the profiling data can be viewed in various ways. One way is to use the "Performance Analysis" perspective in ST32CubeIDE. In this perspective, you can view profiling data in graphical format, such as a call graph or a histogram of function execution times.
5. Analyse and optimize the program: Use the profiling data to identify performance bottlenecks and areas of the program that could be optimized. Make changes to the program code as needed and repeat the profiling process to measure the impact of the changes.

**Why DMA in FreeRTOS?**

Dynamic memory allocation is a feature that allows a program to request memory from the operating system or runtime environment at runtime. In the context of FreeRTOS, dynamic memory allocation is often used to create task stacks, message buffers, and other data structures needed by the operating system.

While dynamic memory allocation can be a useful tool for managing memory in a program, it is true that some coding standards, such as MISRA C, recommend against its use. The main reason for this is that dynamic memory allocation can introduce certain risks and limitations, such as:

1. **Memory fragmentation:** Dynamic memory allocation can cause memory fragmentation, which can make it difficult to allocate large blocks of memory.
2. **Run-time errors**: Dynamic memory allocation can also lead to run-time errors, such as segmentation faults or out-of-memory errors, which can be difficult to diagnose and fix.
3. **Non-deterministic behaviour:** Dynamic memory allocation can lead to non-deterministic behaviour, which can make it difficult to test and verify the correctness of a program.

However, in the case of FreeRTOS, dynamic memory allocation is often used to allocate memory for the creation of tasks and other operating system resources**. In some cases, static memory allocation may not be practical or efficient, especially when dealing with variable-sized data structures or when the maximum number of resources is not known in advance.**

Therefore, while it is true that some coding standards recommend against the use of dynamic memory allocation, it is ultimately up to the developer to determine the best approach for their specific application and requirements. In the case of FreeRTOS, dynamic memory allocation is often a practical and necessary tool for managing memory and resources in real-time embedded systems.

**Preemptive scheduling and cooperative scheduling**

Preemptive scheduling and cooperative scheduling are two different approaches to scheduling tasks or processes in an operating system.

In preemptive scheduling, the operating system can interrupt a running task and allocate CPU time to another task that has a higher priority or is more important. This means that a process can be "preempted" by another process at any time. The preempted process may not have a chance to finish its current task before it is interrupted, and it must save its current state before it is swapped out. Preemptive scheduling can lead to better responsiveness and faster completion times for high-priority tasks, but it may also incur some overhead due to frequent context switches.

In cooperative scheduling, the processes or tasks voluntarily yield control to the operating system after they have completed their current task. The operating system does not forcibly interrupt a running task, and each task must explicitly give up its time slice before another task can execute. Cooperative scheduling is simpler and can be less prone to performance overhead, but it can also lead to problems such as priority inversion or deadlock, where a low-priority task holds a resource that a higher-priority task needs.

Overall, preemptive scheduling is more commonly used in modern operating systems, as it allows for better resource utilization and responsiveness. However, cooperative scheduling may still be used in certain applications, such as real-time systems where predictable performance is more important than maximizing resource utilization

**API in FreeRTOS**

FreeRTOS is a popular open-source real-time operating system for embedded systems. It provides a set of APIs that enable developers to build real-time applications efficiently. Here are some important APIs used in FreeRTOS:

1. Task Management API: This API provides functions to create, delete, suspend, resume, and manage tasks. Tasks are the basic units of execution in FreeRTOS.
2. Queue Management API: This API provides functions to create, delete, and manage message queues. Message queues are used for inter-task communication and synchronization.
3. Semaphore Management API: This API provides functions to create, delete, and manage semaphores. Semaphores are used for task synchronization and mutual exclusion.
4. Mutex Management API: This API provides functions to create, delete, and manage mutexes. Mutexes are used for mutual exclusion between tasks and to prevent concurrent access to shared resources.
5. Event Group Management API: This API provides functions to create, delete, and manage event groups. Event groups are used for task synchronization and to signal events.
6. Timer Management API: This API provides functions to create, delete, and manage timers. Timers are used for scheduling tasks and for periodic events.
7. Memory Management API: This API provides functions to dynamically allocate and deallocate memory from the FreeRTOS heap.
8. Interrupt Management API: This API provides functions to enable and disable interrupts, and to install interrupt service routines (ISRs).

These APIs, along with other supporting functions, provide the necessary building blocks for developing real-time applications on FreeRTOS.

**Important API names used in FreeRTOS**:

* Task Management API:

xTaskCreate()

vTaskDelete()

vTaskSuspend()

vTaskResume()

xTaskDelay()

* Queue Management API:

xQueueCreate()

xQueueSend()

xQueueReceive()

uxQueueMessagesWaiting()

* Semaphore Management API:

xSemaphoreCreateBinary()

xSemaphoreTake()

xSemaphoreGive()

xSemaphoreGiveFromISR()

* Mutex Management API:

xSemaphoreCreateMutex()

xSemaphoreTake()

xSemaphoreGive()

xSemaphoreGiveFromISR()

* Event Group Management API:

xEventGroupCreate()

xEventGroupSetBits()

xEventGroupClearBits()

xEventGroupWaitBits()

* Timer Management API:

xTimerCreate()

xTimerStart()

xTimerStop()

xTimerReset()

* Memory Management API:

pvPortMalloc()

vPortFree()

* Interrupt Management API:

vPortEnterCritical()

vPortExitCritical()

xPortPendSVHandler()

xPortSysTickHandler()

1. **e** prefix: Used for variables that represent enumeration types. For example, **eTaskState** is an enumeration type used to represent the state of a task.
2. **pc** prefix: Used for variables that represent strings (pointer to char). For example, **pcTaskName** is a string that identifies a task.
3. **ux** prefix: Used for variables that represent unsigned integer types. For example, **uxTaskPriority** is an unsigned integer that represents the priority of a task.
4. **px** prefix: Used for variables that represent pointers. For example, **pxQueueHandle** is a pointer to a queue.
5. **v** prefix is often used for variables that represent void functions, such as **vTaskFunction()**,The **v** prefix stands for "void", which is the return type of the function.
6. **x** prefix - represent handles, such as **xTaskHandle** , The **x** prefix stands for "handle", which indicates that the variable holds a reference to an object or resource managed by the FreeRTOS kernel.
7. **pv** prefix is also used in FreeRTOS naming conventions, and it stands for "pointer to void".
8. **us** prefix is not a commonly used prefix in FreeRTOS naming conventions. However, it is sometimes used to indicate variables that represent unsigned short integer types.

**handle**

* In FreeRTOS API, a "handle" typically refers **to an opaque data type that represents an instance of a FreeRTOS object such as a task, a queue, a semaphore, or a mutex**. A handle is used as an identifier to refer to a specific instance of the object and to perform operations on it.
* For example, when you create a task using the xTaskCreate() function in FreeRTOS, it returns a TaskHandle\_t type handle that you can use to refer to that task instance in subsequent operations like suspending, resuming or deleting the task.
* In general, a handle provides a level of abstraction that allows you to manipulate FreeRTOS objects without having to know their internal implementation details. By using handles, FreeRTOS provides a consistent way to interact with different types of objects and allows you to write more modular and reusable code.

**"pvTaskCode" and "pxCreatedTask"(Arguments of xtastcreate()function)**

In FreeRTOS, "pvTaskCode" and "pxCreatedTask" are both parameters used when creating a task using the xTaskCreate() function.

**"pvTaskCode"** is a pointer to the function that contains the code for the task being created. This function should be of type "TaskFunction\_t", and takes a single argument of type "void \*". The function should contain the code that the task will execute when it is started.

Example:

void vTaskFunction(void \*pvParameters) {

// Code for the task goes here

}

xTaskCreate(vTaskFunction, "Task Name", 1000, NULL, 1, NULL);

In the above example, "vTaskFunction" is the name of the function that contains the code for the task. It takes a single argument of type "void \*", which can be used to pass parameters to the task. The "xTaskCreate()" function creates the task and specifies the function to execute as the first parameter.

**"pxCreatedTask"** is a pointer to a variable that will be set to the task handle when the task is created. The task handle is a reference to the created task and is used in other FreeRTOS API functions to perform operations on the task.

Example:

void vTaskFunction(void \*pvParameters)

{

// Code for the task }

TaskHandle\_t xTaskHandle;

xTaskCreate(vTaskFunction, "Task Name", 1000, NULL, 1, &xTaskHandle);

In the above example, "xTaskHandle" is a variable of type "TaskHandle\_t" that will be set to the handle of the task when it is created. The "xTaskCreate()" function creates the task and sets the task handle to "xTaskHandle".

So "pvTaskCode" is used to **specify the function that contains the code for the task**, **while "pxCreatedTask" is used to obtain a handle to the created task**.

**vTaskDelay()**

* **vTaskDelay()** is a FreeRTOS function used in embedded systems programming with Real-Time Operating Systems (RTOS). It is **used to suspend the execution of the current task for a specific period of time, allowing other tasks in the system to execute.**
* The function takes a single argument, which is the number of ticks to delay for**. A tick is a unit of time defined by the RTOS, and the duration of a tick is configurable.** For example, if the tick rate is set to 1 millisecond and the argument to vTaskDelay() is 1000, the task will be delayed for one second.
* During the delay period, the task is blocked and does not consume any CPU resources. When the delay period is over, the task becomes eligible to run again and will be scheduled by the RTOS.
* It is important to note that vTaskDelay() **should not be used in tasks that have a real-time requirement**, as the actual delay period can be longer than the requested period due to the scheduling of other tasks and interrupts. In such cases, it is recommended to use a timer or an interrupt to ensure accurate timing.
* **vTaskDelay() follows cooperative scheduling**, which is a scheduling algorithm used by many RTOSs including FreeRTOS. In cooperative scheduling, the currently running task voluntarily yields the CPU to allow other tasks to run, typically by calling a blocking function such as vTaskDelay(). This allows the RTOS to manage the system resources efficiently and ensures that all tasks get a fair share of the CPU time.

**Queue –**

* The maximum no. of message in message queue is determined by the size of the message and available memory.
* Message Queues doesn’t have priorities
* The starvation problem in using FreeRTOS queue occurs when a task with a lower priority is continuously blocked on a queue, preventing higher-priority tasks from executing.
* This can happen when a task that is waiting to send data to a queue (using xQueueSend) is preempted by a higher-priority task that is waiting to receive data from the same queue (using xQueueReceive). If the receiving task never gets blocked and continuously runs, the sending task will never get a chance to execute and send its data, causing a deadlock situation.

**Creating a Queue:**

To create a queue in C, we use the "queue.h" header file and declare a struct of type "Queue". We then use the "xQueueCreate()" function to create a queue, which takes two arguments: the maximum number of elements that the queue can hold, and the size of each element in bytes.

Example:

#include <stdio.h>

#include "queue.h"

QueueHandle\_t myQueue;

myQueue = xQueueCreate(5, sizeof(int));

The above code creates a queue called "myQueue" that can hold a maximum of 5 elements of type "int".

**Adding elements to a Queue:**

To add an element to a queue in C, we use the "xQueueSend()" function, which takes three arguments: the queue handle, a pointer to the data being sent, and a timeout value in ticks.

Example:

int data = 10;

xQueueSend(myQueue, &data, portMAX\_DELAY);

The above code adds an integer value of 10 to the end of the queue called "myQueue". The "portMAX\_DELAY" argument causes the function to block indefinitely if the queue is full.

**Deleting elements from a Queue:**

To delete an element from a queue in C, we use the "xQueueReceive()" function, which takes three arguments: the queue handle, a pointer to a buffer to store the received data, and a timeout value in ticks.

Example:

int received\_data;

xQueueReceive(myQueue, &received\_data, portMAX\_DELAY);

The above code removes the first element from the queue called "myQueue" and stores it in the "received\_data" variable. The "portMAX\_DELAY" argument causes the function to block indefinitely if the queue is empty.