RTOS Introduction

What is RTOS?
A Real-Time Operating System(RTOS) is an operating system that is intended to fulfill the requirement of real-time application. It is able to process data as comes in, typically without buffering delays.

What makes RTOS different from the Simple C program?
The key factors in Real Time Operating Systems are minimum interrupt latency and minimum thread switching latency. The Real Time Operating System is valued more for how quickly and how predictably it responds to complete the tasks in a given period of time.

What are the different types of RTOS?

There are three types of RTOS:
1. Hard RTOS; bound to complete a task within a given deadline
2. Firm RTOS; bound by a deadline but if they miss the deadline, it is acceptable but not in the case of Hard RTOS.
3. Soft RTOS; not bound by any deadline.

VxWorks, FreeRTOS,OSEK/AUTOSAR OS, QNX – These are some of the famous RTOS available in the market

FreeRTOS is always beneficial for the reasons listed below:

- Abstract out timing information
- Maintainability/Extensibility
- Modularity
- Cleaner interfaces
- Easier testing (in some cases)
- Code reuse
- Improved efficiency
- Idle time
- Flexible interrupt handling
- Mixed processing requirements
- Easier control over peripherals

Here are some disadvantages of RTOS:

- Low Priority Tasks
- Precision of code
- Limited Tasks
- Complex Algorithms
- Device driver and interrupt signals
- Thread Priority
- Expensive
- Not easy to program

Introduction to FreeRTOS

Unlike typical real-time operating systems, FreeRTOS is specially designed for microcontrollers. Because Microcontrollers come with limited resources, therefore, we need an operating system as per the available resources of microcontrollers. It is an open-source Kernel which means it can download free of cost and be used in RTOS-based applications. Although, it has two commercial variants also such as OpenRTOS and SafeRTOS.

As we mentioned earlier, by using FreeRTOS, we can make sure that each task of Arduino will have a deterministic execution pattern and that every task will meet its execution deadline. In other words, it is a scheduler that assigns Arduino CPU resources to every task according to a scheduling algorithm.

The followings are the main kernel features:

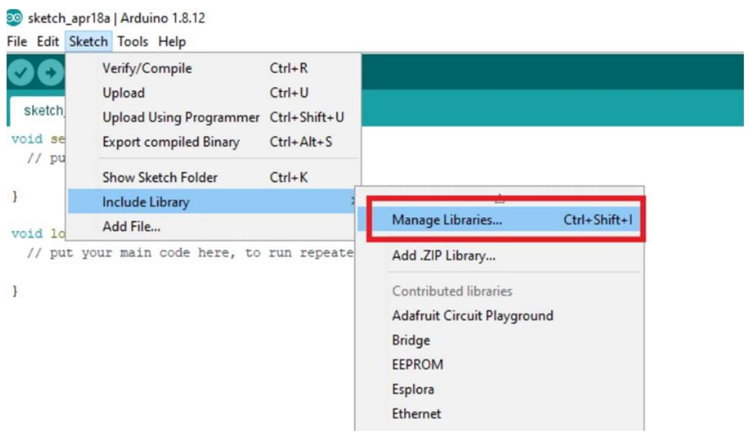
- Fixed Priority Pre-emptive Time slicing Scheduler
- Inter-Process Communication through Queues
- Tasks Management such as Task priority setting, task suspension, task deletion, and Task delay
- Tasks Synchronization with Gatekeeper Tasks, Semaphore, and Mutex
- Timing measurement hook
- Run time Tracing hooks
- Software timer management
- Interrupt management

Using FreeRTOS with Arduino

FreeRTOS is an open-source popular Real-Time Operating System kernel. Furthermore, it is used for embedded devices which as microcontrollers, and Arduino. It is mostly written in C but some functions are written in assembly.

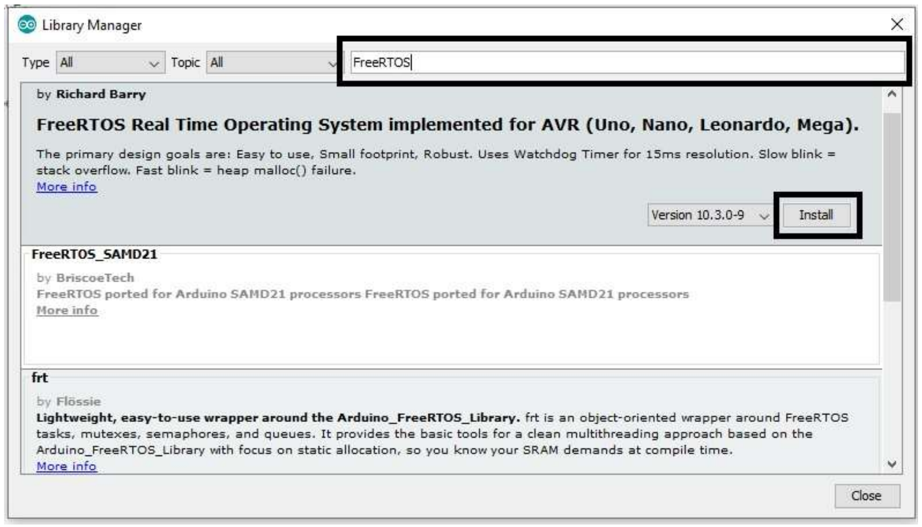
Setting up FreeRTOS with Arduino

1. Download Arduino IDE as a standalone pack(zip pack) from the below link: https://www.arduino.cc/en/software
2. Install FreeRTOS from Arduino Library manager


sketch_apr18a | Arduino 1.8.12
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...
Manage Libraries... Ctrl+Shift+I
Add .ZIP Library...
Contributed libraries
Adafruit Circuit Playground
Bridge
EEPROM
Esplora
Ethernet

```
void se
  // pu
}

void lo
  // put your main code here, to run repeate
}
```


Library Manager

Type All Topic All FreeRTOS

by **Richard Barry**

FreeRTOS Real Time Operating System implemented for AVR (Uno, Nano, Leonardo, Mega).

The primary design goals are: Easy to use, Small footprint, Robust. Uses Watchdog Timer for 15ms resolution. Slow blink = stack overflow. Fast blink = heap malloc() failure.
More info

Version 10.3.0-9 Install

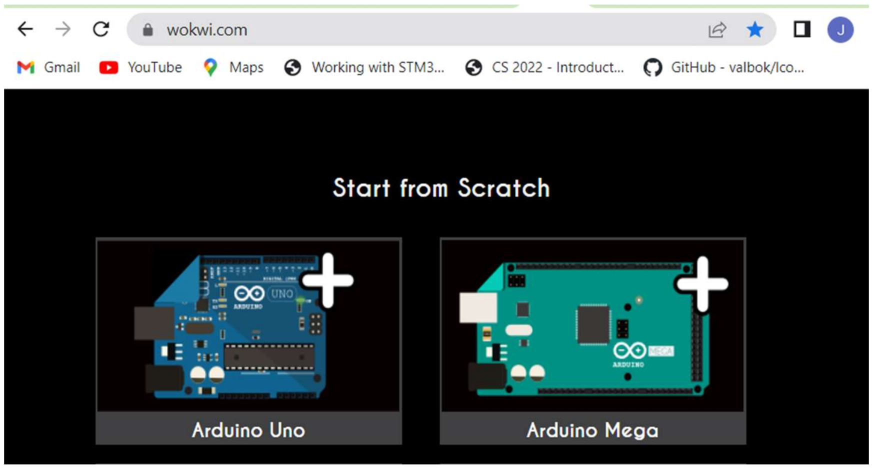
FreeRTOS_SAMD21
by BriscoeTech
FreeRTOS ported for Arduino SAMD21 processors FreeRTOS ported for Arduino SAMD21 processors
More info

frt
by Flössie
Lightweight, easy-to-use wrapper around the Arduino_FreeRTOS_Library. frt is an object-oriented wrapper around FreeRTOS tasks, mutexes, semaphores, and queues. It provides the basic tools for a clean multithreading approach based on the Arduino_FreeRTOS_Library with focus on static allocation, so you know your SRAM demands at compile time.
More info

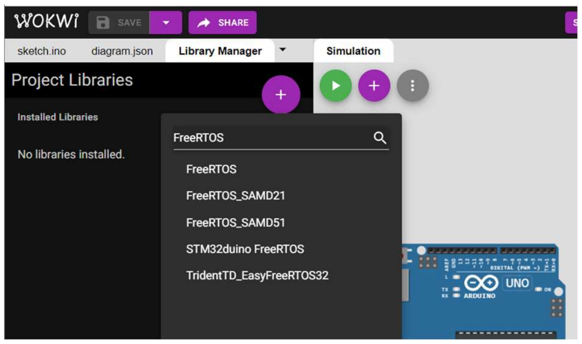
Close

Setting up FreeRTOS in Simulator

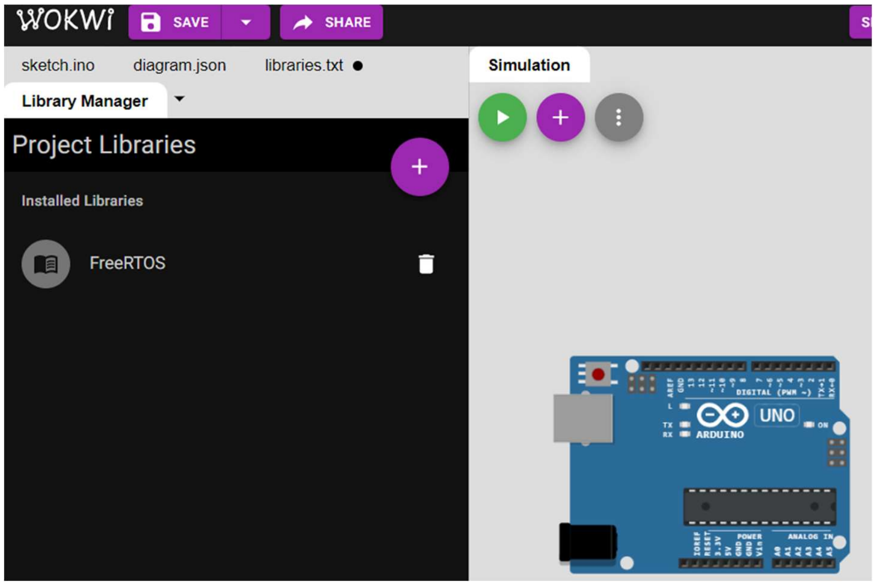
1. Add Arduino Mega/Uno in wokwi simulator (under start from scratch section) in the below link:
   https://wokwi.com/



2. Add FreeRTOS library under Library Manager by click + icon



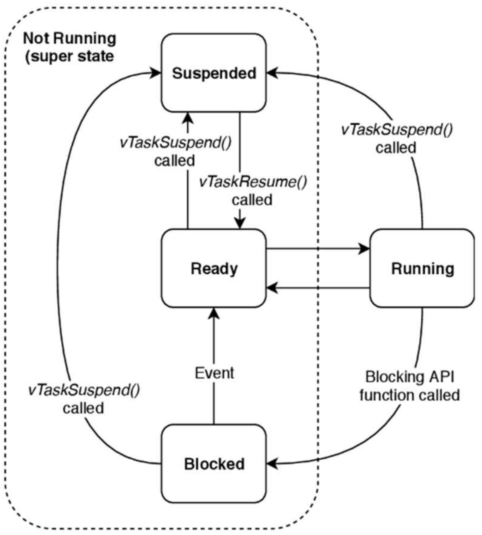
3. Add FreeRTOS package

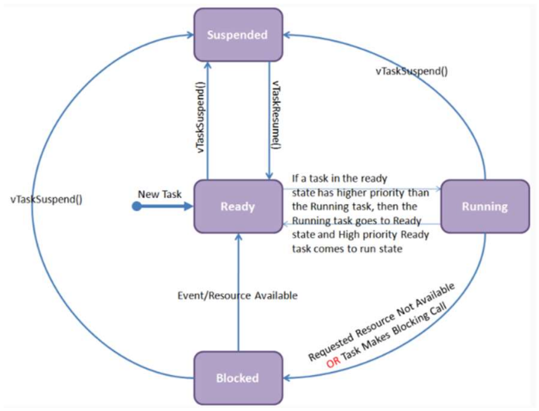


FreeRTOS Task Management

In a multitasking system, an application can consist of many tasks. If we are using a single-core processor, then only one task can run on the processor at any given time. Hence, only one task will be in the running state and all other tasks will be in the not running state. That means, in RTOS based applications, tasks can be either in running state or not running state.

The running state of a task can be further divided into three sub-states such as blocked state, ready state, and suspended state. The figure below shows the transition lifetime of a task in a multitasking system.




Suspended
vTaskSuspend()
vTaskSuspend()
vTaskResume()
vTaskSuspend()
New Task
Ready
If a task in the ready state has higher priority than the Running task, then the Running task goes to Ready state and High priority Ready task comes to run state
Running
Event/Resource Available
Requested Resource Not Available
OR Task Makes Blocking Call
Blocked

FreeRTOS Tasks States

Blocked State

A task can be in a blocked state due to many reasons. For example, a task is delayed due to the periodic nature of the task and it will become available periodically after every specified delay. Another reason could be due to interrupt waiting or resource waiting. A task is waiting for the external interrupt or a resource such as binary semaphore, counting semaphore, and a mutex.

Suspended State

A suspended state is also a substate of a not-running task. The application programmer can suspend the task by using vTaskSuspend() FreeRTOS API function and resume the task using vTaskResume() API function.

The tasks in blocked or suspended states cannot be scheduled by a scheduler.

Ready State

If a task is not in either a blocked or suspended state, it will be in a ready state. Tasks that are in a ready state, are ready for execution as soon as the processor picks them according to a scheduling policy. Whenever a task makes the transition from a running state to a not-running state, a context switch happens. That means, it saves its registers values, temporary variable values, etc into its task control block, and next time when it will enter the running state again, it will start execution from the same point where it left the execution. To start execution from the same location, it will again load values from TCB into processor registers.

Creating Task in FreeRTOS

Multitasking benefits can be achieved only by performing every execution with the help of tasks. For instance, in your ATMega project, you want to measure temperature and want to display its value on LCD. We can divide this project into three tasks such as:

1. Read Arduino ADC Value
2. Converter ADC Value into Temperature
3. Print Temperature on LCD

In this way, we can also assign, priority to each task according to the critical nature of the task. For example, ADC value reading is the most important, and printing temperature value on LCD is the least important task. Hence, we can assign the highest priority to the ADC task and the lowest to the LCD task.

API For Task creation in FreeRTOS

xCreateTask() function is used to create tasks and adds them to the ready queue. It takes 5 arguments as inputs to define various features of the task.

Ex: `xTaskCreate(MyTask_pointer, "task_name", 100, Parameter, Priority, TaskHandle);`

Argument List:

1. **MyTask_pointer**: This first argument to task creation function is a pointer to a function definition of a task. Because we need to define a task with the help function. We will see later how to define a function and pass this pointer to a function as an argument.
2. **task_name**: This argument is just the name of the function/task that we will create.
3. **StackDepth**: In multitasking, each task/thread has its own stack. It defines the stack size of a task in bytes. Therefore, you should make sure to select stack size according to the complexity of the computation. For instance, we select 100 bytes size in this example.
4. **Parameter**: 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. This argument is a pointer to a variable that the task (function) can receive.
5. **Priority:** This is an important parameter because it is used to set the priority of tasks. We set priority with passing numbers as an argument. 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.
6. **TaskHandle:** 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.

Example 1

FreeRTOS follows both pre-emptive scheduling and cooperating scheduling. But by default, this API implements pre-emptive time-slicing scheduling. That means high priority tasks pre-empt low priority tasks and equal priority tasks use time-shared policy to get CPU resources. This code creates four tasks with different priorities. But all three tasks are periodic. Because of vTaskDelay() function, each task goes to a blocking state for a specified time.

```
#include <Arduino_FreeRTOS.h>
void setup()
//Initialize the Serial Monitor with 9600 baud rate
{
Serial.begin(9600);
Serial.println(F("In Setup function")); //use of F saves memory
//Set the digital pins 8 to 11 as digital output pins
  pinMode(8,OUTPUT);
  pinMode(9,OUTPUT);
  pinMode(10,OUTPUT);
  pinMode(11,OUTPUT);

//Create three tasks with labels Task1, Task2 and Task3 and assign the
priority as 1, 2 and 3 respectively.
//We also create the fourth task labeled as IdelTask when there is no
task in
//operation and it has the highest priority.

 xTaskCreate(MyTask1, "Task1", 100, NULL, 1, NULL);
 xTaskCreate(MyTask2, "Task2", 100, NULL, 2, NULL);
 xTaskCreate(MyTask3, "Task3", 100, NULL, 3, NULL);
 xTaskCreate(MyIdleTask, "IdleTask", 100, NULL, 0, NULL);}
```

```
//We can change the priority of task according to our desire by
changing the numeric’s //between NULL texts.

void loop()

{
//There is no instruction in the loop section of the code.
// Because each task executes on interrupt after specified time
}

//The following function is Task1. We display the task label on Serial
monitor.

static void MyTask1(void* pvParameters)
{

  while(1)

  {
    digitalWrite(8,HIGH);
    digitalWrite(9,LOW);
    digitalWrite(10,LOW);
    digitalWrite(11,LOW);
    Serial.println(F("Task1"));
    vTaskDelay(100/portTICK_PERIOD_MS);
  }
}

//Similarly this is task 2

static void MyTask2(void* pvParameters)

{
while(1)

  { digitalWrite(8,LOW);
    digitalWrite(9,HIGH);
    digitalWrite(10,LOW);
    digitalWrite(11,LOW);
    Serial.println(F("Task2"));
    vTaskDelay(110/portTICK_PERIOD_MS);
  }
}

//Similarly this is task 3

static void MyTask3(void* pvParameters)
{
while(1)
  {
   digitalWrite(8,LOW);
```

```
    digitalWrite(9,LOW);
    digitalWrite(10,HIGH);
    digitalWrite(11,LOW);
    Serial.println(F("Task3"));
    vTaskDelay(120/portTICK_PERIOD_MS);
  }
}

//This is the idle task which has the lowest priority and calls when no
task is running.

static void MyIdleTask(void* pvParameters)

{
  while(1)
   {
    digitalWrite(8,LOW);
    digitalWrite(9,LOW);
    digitalWrite(10,LOW);
    digitalWrite(11,HIGH);
    Serial.println(F("Idle state"));
    delay(50);
  }
}
```

OUTPUT

As you can see from the output of the serial monitor, the lowest priority task (Idle task) executes only when the processor is free and not any other task is available to execute.

T1: Task3 starts to execute first being the highest priority task that is three. After that, it enters the blocking state for 120ms.

T2: After that Task2 starts to execute because it now attains the highest priority because Task3 is in a blocking state. Similarly, it also completes its execution and goes to a blocked state, and remains there for 110ms.

T3: Currently, both Task3 and Task2 are waiting for the blocked time to end. Hence, Task1 starts to execute because its priority is higher than the idle task. Therefore, it also enters the running state and completes its execution, and goes to blocking mode for 100ms.

T4: Now, all high priority task has completed their execution and they are in blocked state waiting for their blocking time to end. Therefore, the processor keeps executing, idle_task until a high priority task comes out of a blocking state.

Furthermore, the shortest blocking time is used for task1 that is 100ms. Therefore, it will enter the running state before other high priority tasks and similarly for task2 and task3.

After that, all threads keep executing according to their priority and blocking time.

WokWi logic analyzers record the samples on .a vcd file which can be opened using the Pulseview tool (can be downloaded using the below link):

https://sigrok.org/wiki/Downloads

Exercise 1: Compare it with the real HW output and use real logic analyser

Before starting with RTOS working, let's see what is a Task. The task is a piece of code that is schedulable on the CPU to execute. So, if you want to perform some task, then it should be scheduled using kernel delay or using interrupts. This work is done by the Scheduler present in the kernel. In a single-core processor, the scheduler helps tasks to execute in a particular time slice but it seems like different tasks are executing simultaneously. Every task runs according to the priority given to it.

Now, let's see what happens in the RTOS kernel if we want to create a task for LED blinking with a one-second interval and put this task on the highest priority.

Ex 2: Implement a counter and 2 LED blinking as 3 separate Task

https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial1-creating-freertos-task-to-blink-led-in-arduino-uno

Frequently used terms in RTOS

1. Task: It is a piece of code that is schedulable on the CPU to execute.

2. Scheduler: It is responsible for selecting a task from the ready state list to the running state. Schedulers are often implemented so they keep all computer resources busy (as in load balancing).

3. Pre-emption: It is the act of temporarily interrupting an already executing task with the intention of removing it from the running state without its cooperation

4. Context Switching: In priority-based pre-emption, the scheduler compares the priority of running tasks with the priority of the ready task list on every systick interrupt. If there is any task in the list whose priority is higher than the running task then a context switch occurs. Basically, in this process contents of different tasks get saved in their respective stack memory.

5. Types of Scheduling policies:

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

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

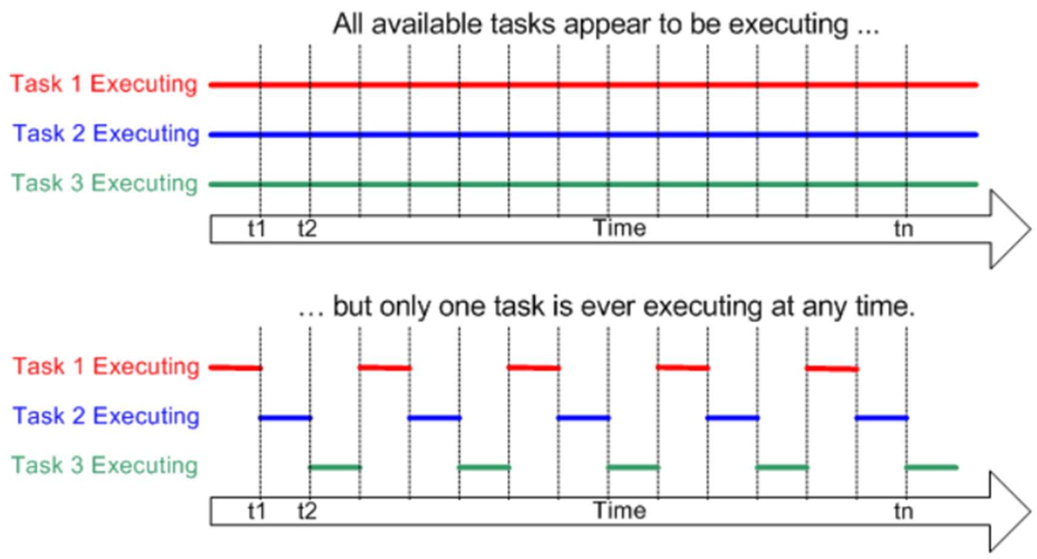
Co-operative Scheduling: Context switching will happen only with the cooperation of running tasks. Task will run continuously until task yield is called.

6.Kernel Objects: For signaling the task to perform some work, the synchronization process is used. To perform this process Kernel objects are used. Some Kernel objects are Events, Semaphores, Queues, Mutex, Mailboxes, etc. We will see how to use these objects in upcoming tutorials.

The kernel is the core component of the operating system. Operating systems like Linux use a kernel, allowing users to appear to access the computer at the same time. Each program being executed is a task (or thread) under the control of the operating system. If an operating system can perform multiple tasks in this way, it can be said to be multitasking.

Multitasking

Traditional processors can only execute one task at a time, but a multitasking operating system can make each task appear to be executed simultaneously by quickly switching between tasks. The following figure shows the relationship between the execution mode of three tasks and time.

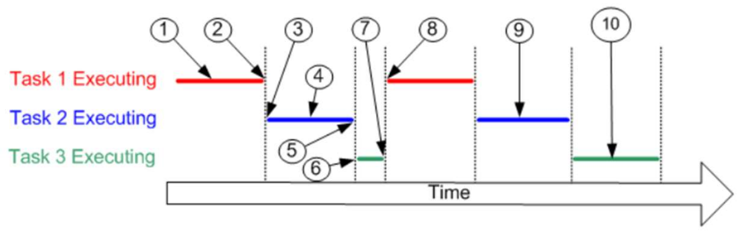


Scheduling

The scheduler is the part of the kernel that is responsible for deciding which task to execute at any particular time. The kernel can pause and resume a task multiple times during the life cycle of the task.

A scheduling strategy is an algorithm used by the scheduler to decide which task to perform at any point in time. A (non-real-time) multi-user system strategy is likely to give each task a "fair" proportion of processor time.

In addition to being involuntarily suspended by the kernel, a task can also choose to suspend itself. If it wants to delay (sleep) for a period of time, or wait (block) for a resource to be available (such as a serial port) or an event (such as a key press), it will do so.


1
2
3
4
5
6
7
8
9
10
Task 1 Executing
Task 2 Executing
Task 3 Executing
Time