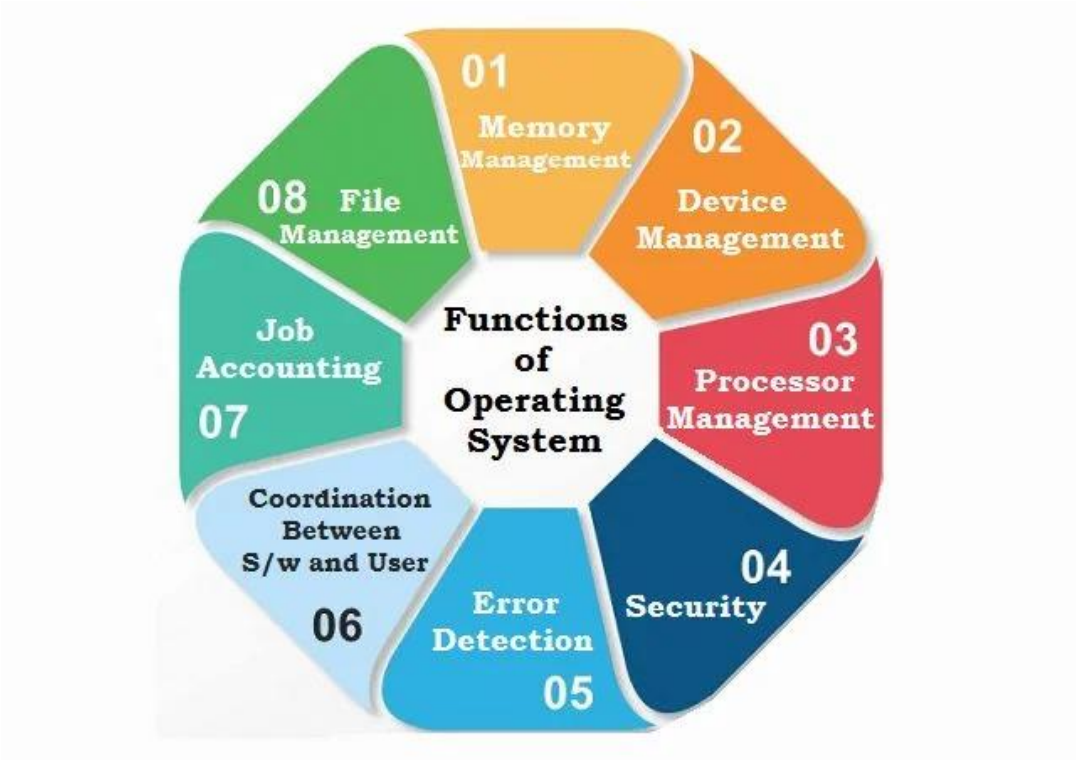
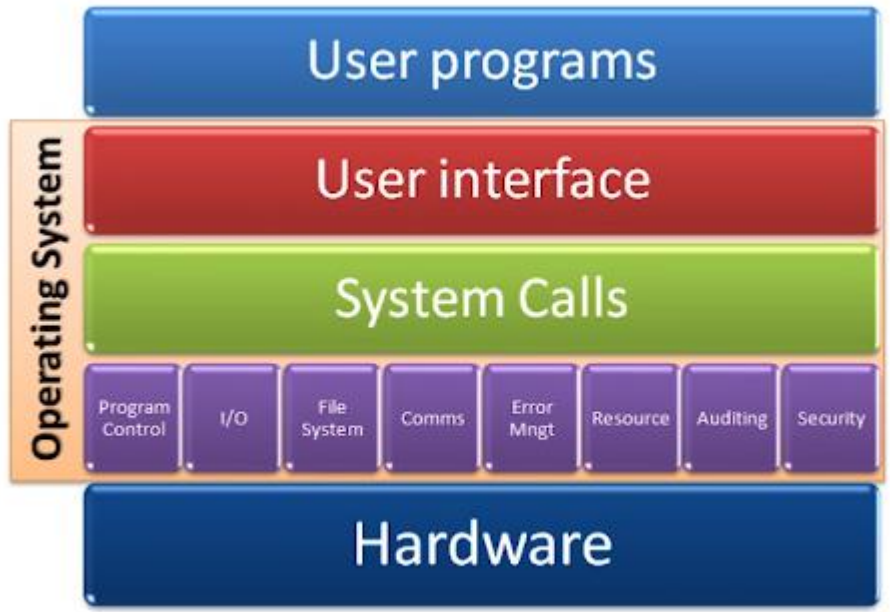
Real Time Operating System (RTOS)

What is Operating System?

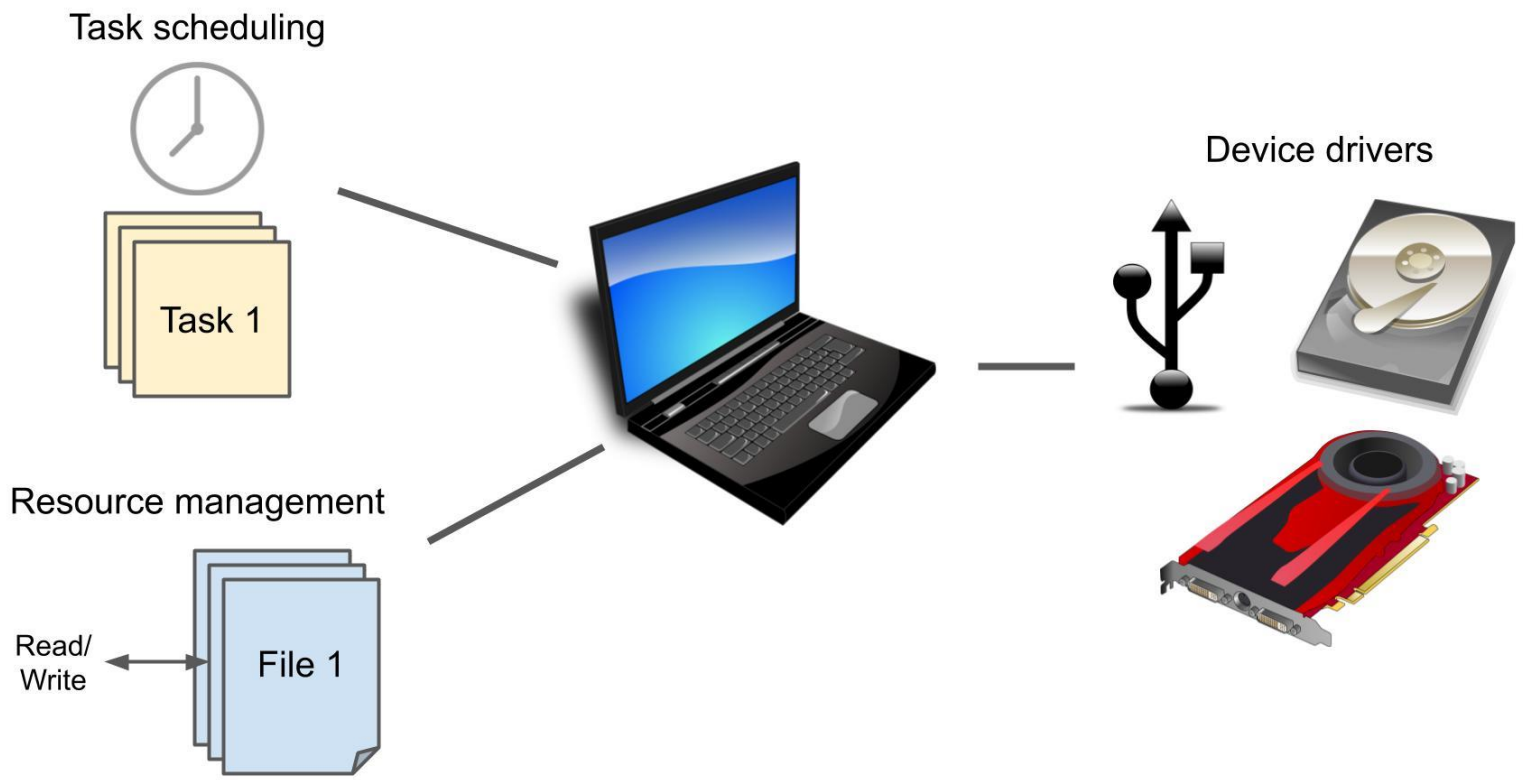
An operating system is system software that manages computer hardware and software resources, and provides common services for computer programs.

OS


Functions of Operating System
01 Memory Management
02 Device Management
03 Processor Management
04 Security
05 Error Detection
06 Coordination Between S/w and User
07 Job Accounting
08 File Management


User programs
Operating System
User interface
System Calls
Program Control
I/O
File System
Comms
Error Mngt
Resource
Auditing
Security
Hardware

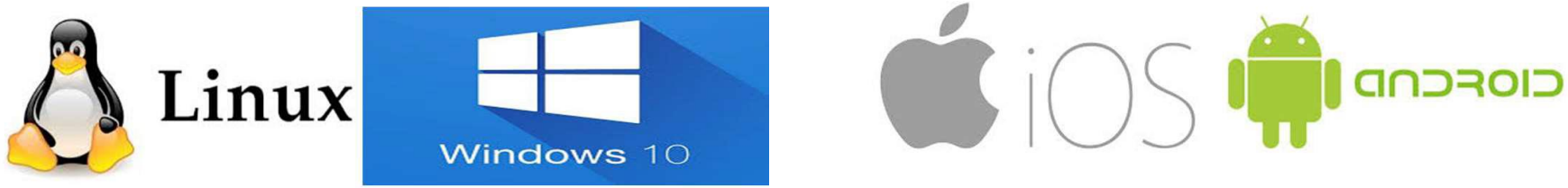
General Purpose Operating System (GPOS)



A general-purpose OS represents an array of operating systems intended to run a multitude of applications on a broad selection of hardware, enabling a user to run one or more applications or tasks simultaneously

GPOS Vs RTOS

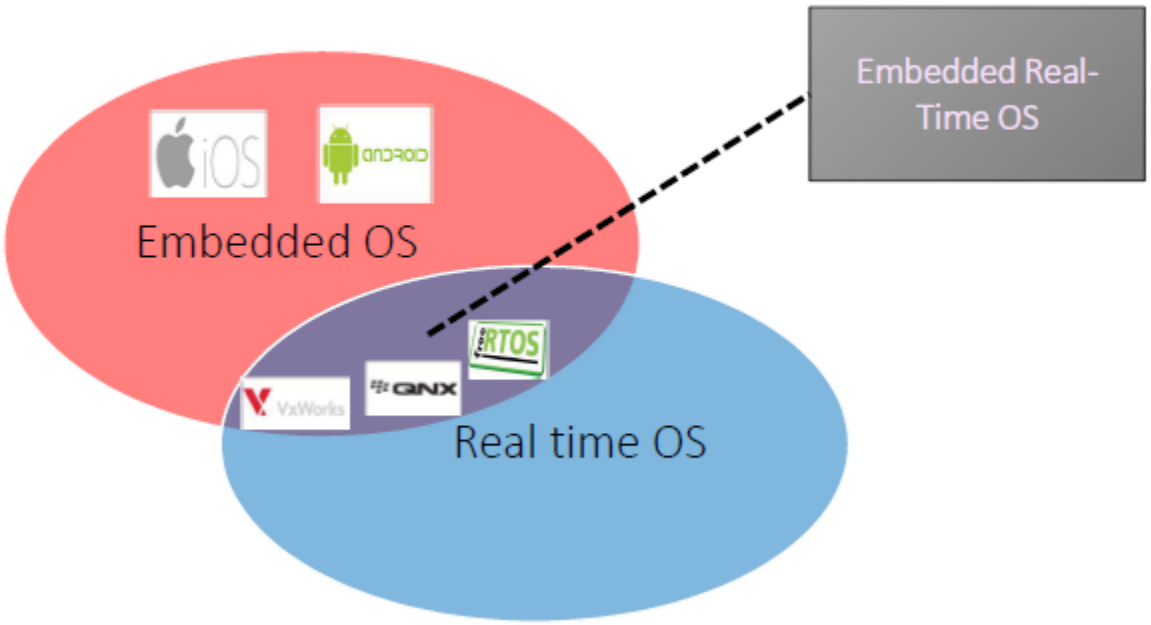
GPOS



RTOS

OS for Embedded System

- FreeRTOS
- Mbed OS
- Mynewt OS
- Zephyr
- RIOT
- TinyOS
- Contiki OS



RTOS vs GPOS Task Scheduling

GPOS

In the case of a GPOS – task scheduling is not based on “priority” always!

RTOS

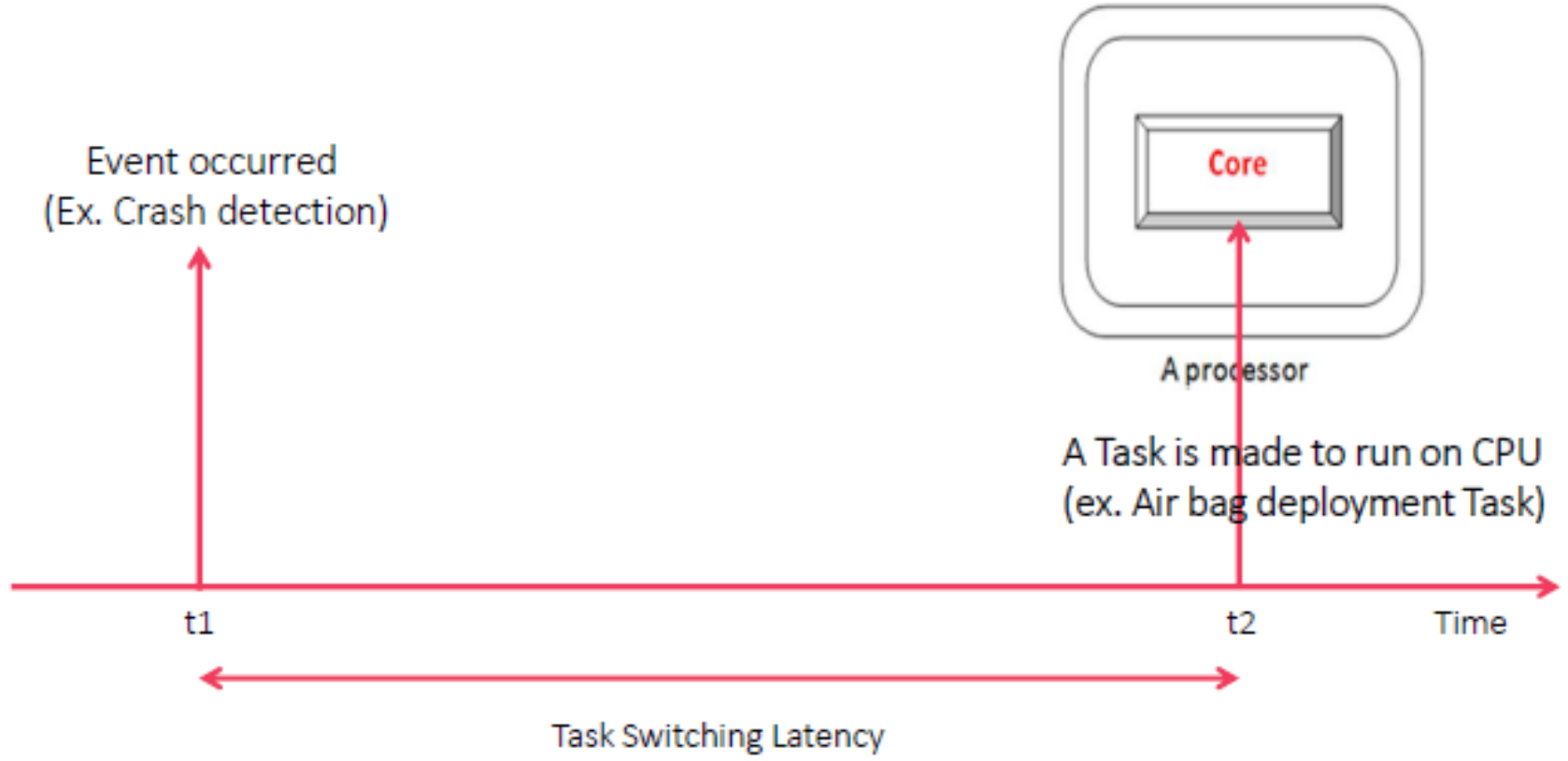
Where as in RTOS – scheduling is always priority based. Most RTOS use pre-emptive task scheduling method which is based on priority levels.

RTOS vs GPOS Task Switching Latency

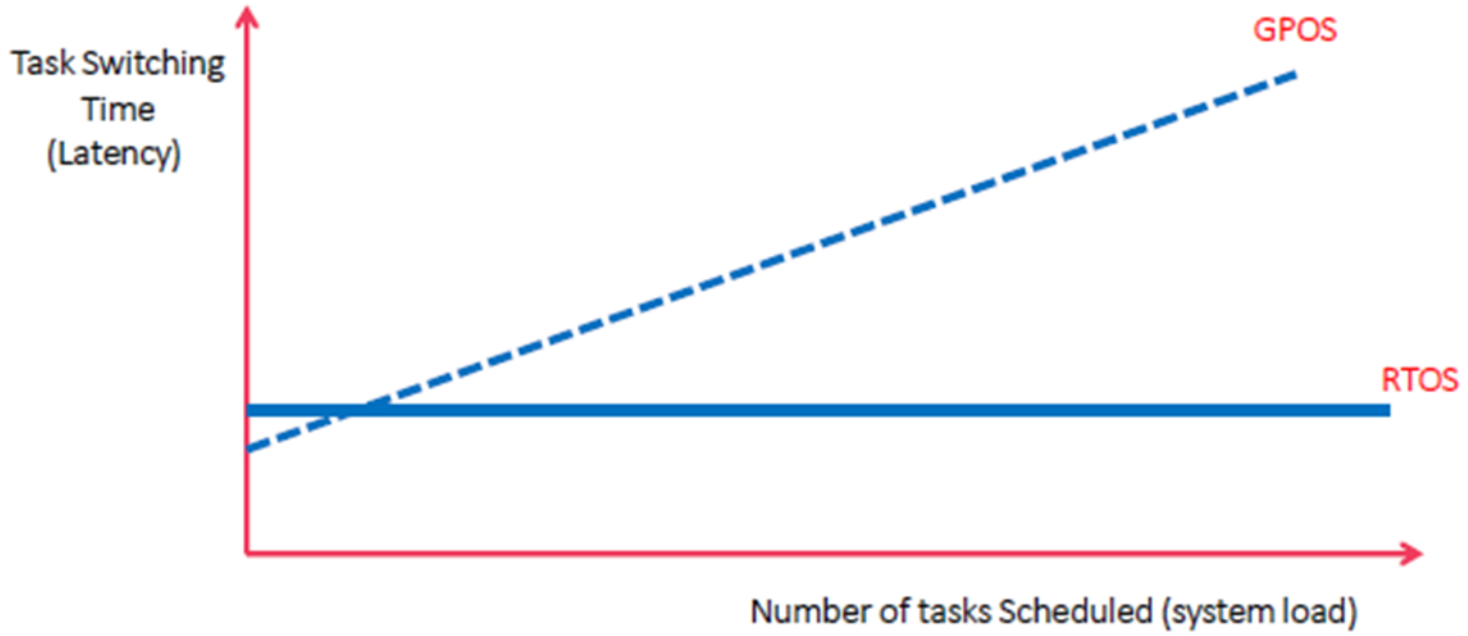
In Computing , Latency means “The time that elapses between a stimulus and the response to it”.

Task switching latency means, that time gap between “A triggering of an event and the time at which the task which takes care of that event is allowed to run on the CPU”

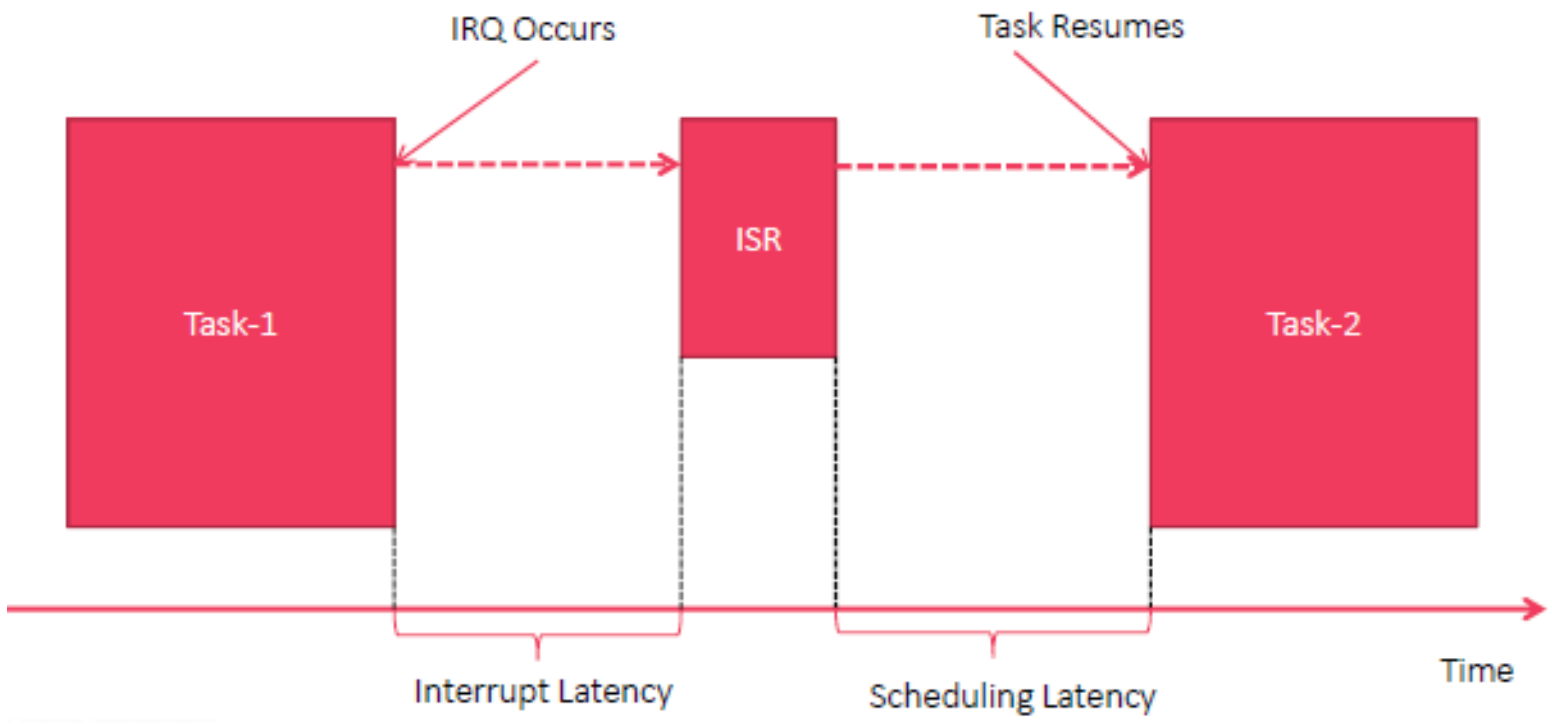
RTOS vs GPOS Task Switching Latency



RTOS vs GPOS Task Switching Latency



RTOS vs GPOS: Interrupt Latency



RTOS vs GPOS: Priority Inversion

Priority inversion effects are in-significant

Priority inversion effects must be solved

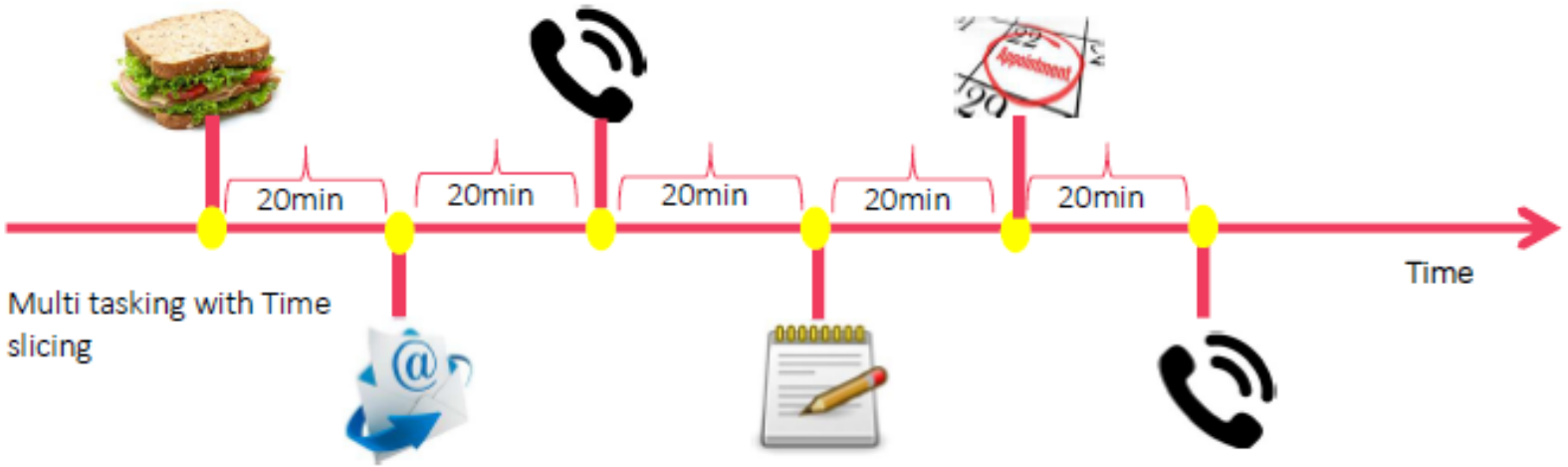
RTOS vs GPOS: Kernel Preemption

Preemption
In computing preemption is the act of temporarily removing the task from the running state without its co-operation

In RTOS , Threads execute in order of their priority. If a high-priority thread becomes ready to run, it will, within a small and bounded time interval, take over the CPU from any lower-priority thread that may be executing that we call preemption. The lower priority task will be made to leave CPU, if higher priority task wants to execute.

The kernel operations of a RTOS are pre-emptible where as the kernel operations of a GPOS are not pre-emptible

What is Multitasking?



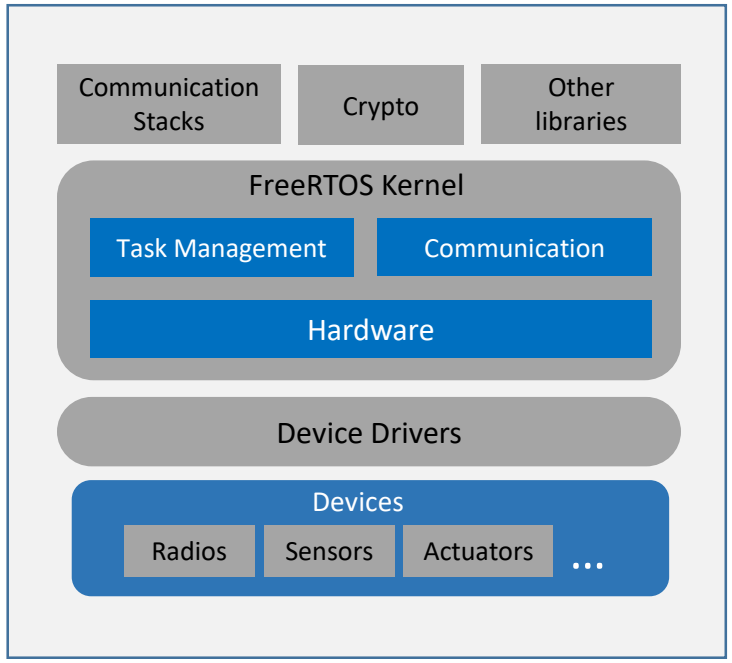
You	Assistant-1	Assistant-2	Assistant-3
Having Breakfast	Emailing on behalf of you	Receiving calls on behalf of you	Managing appointments

FreeRTOS

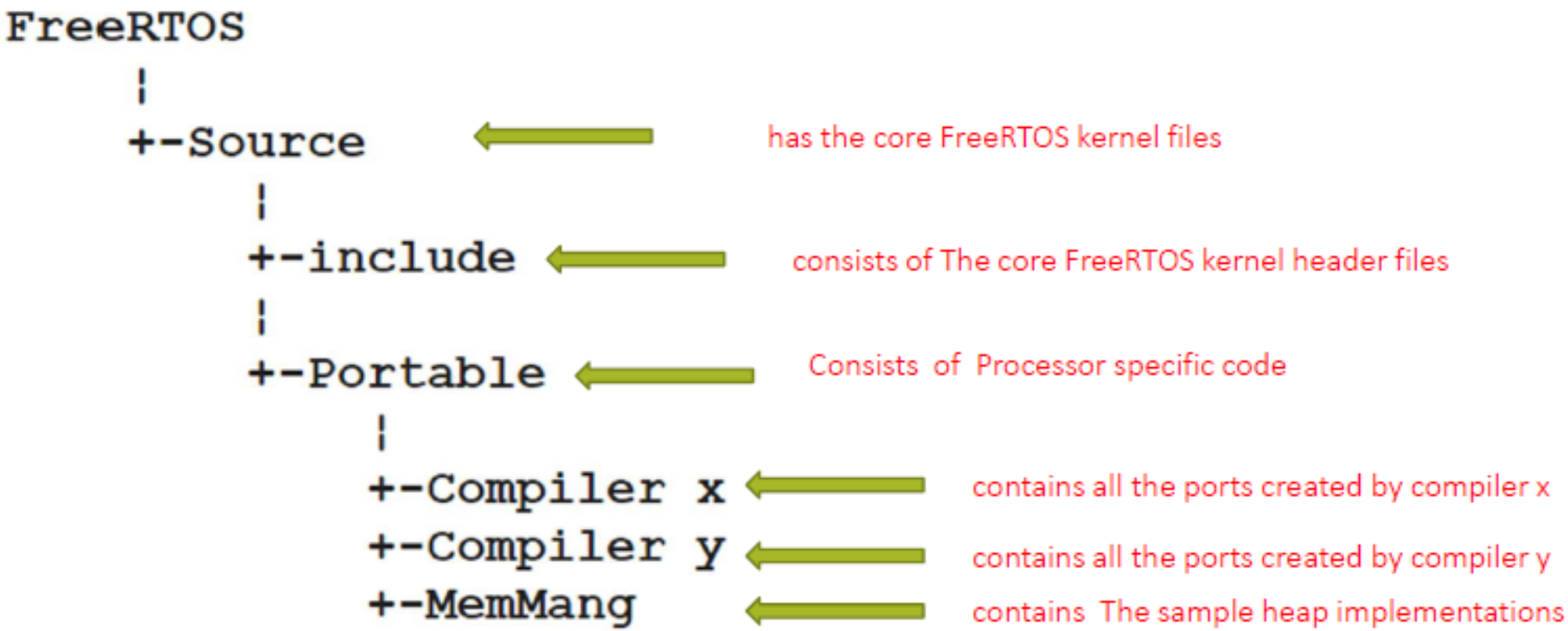
- FreeRTOS is a RTOS kernel created in 2003, broadly used In MCU based projects
- Now distributed under the MIT licence.

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

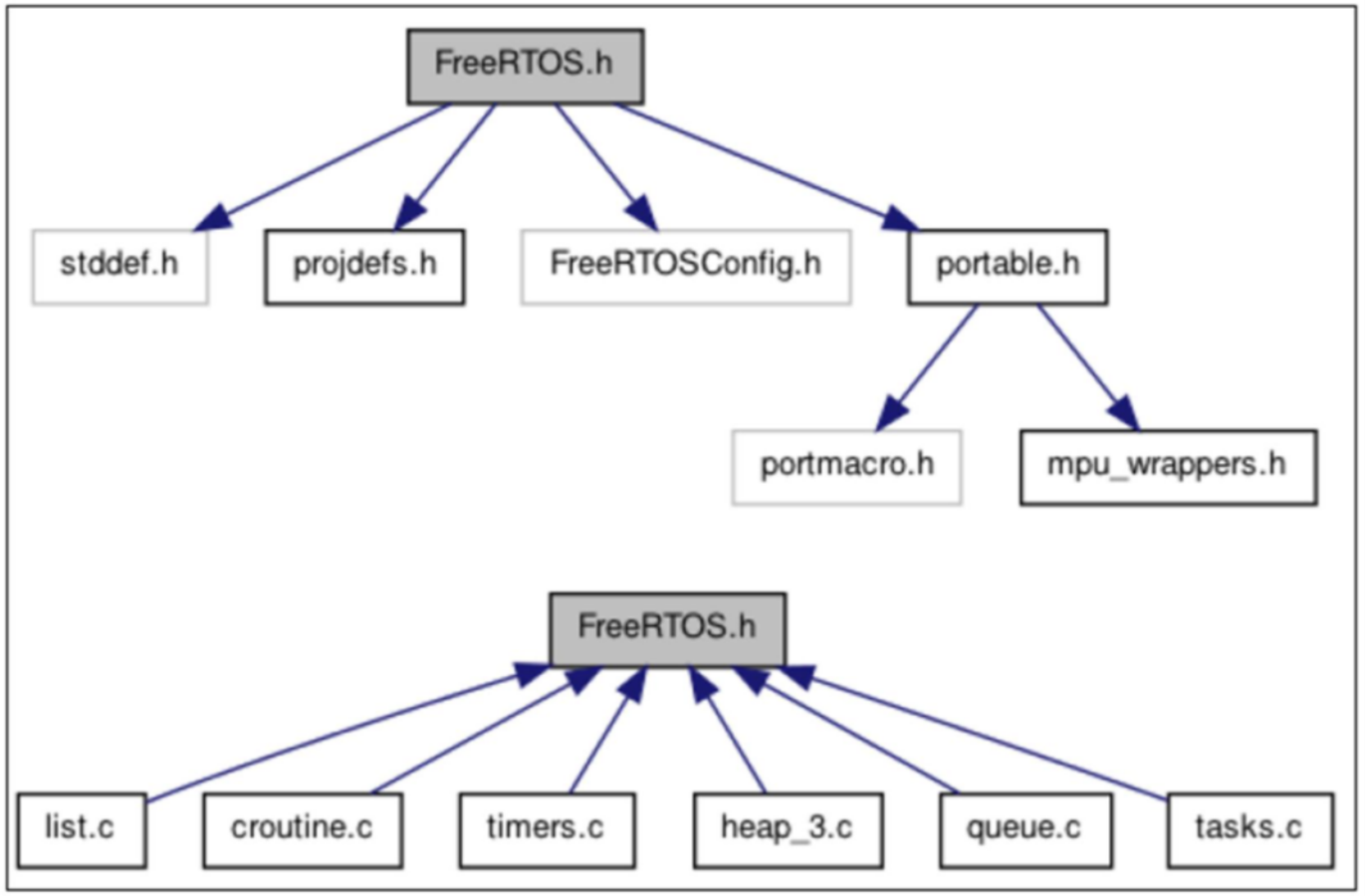
Download FreeRTOS: https://freertos.org/a00104.html
Reference Book FreeRTOS: https://freertos.org/Documentation/RTOS_book.html



FreeRTOS

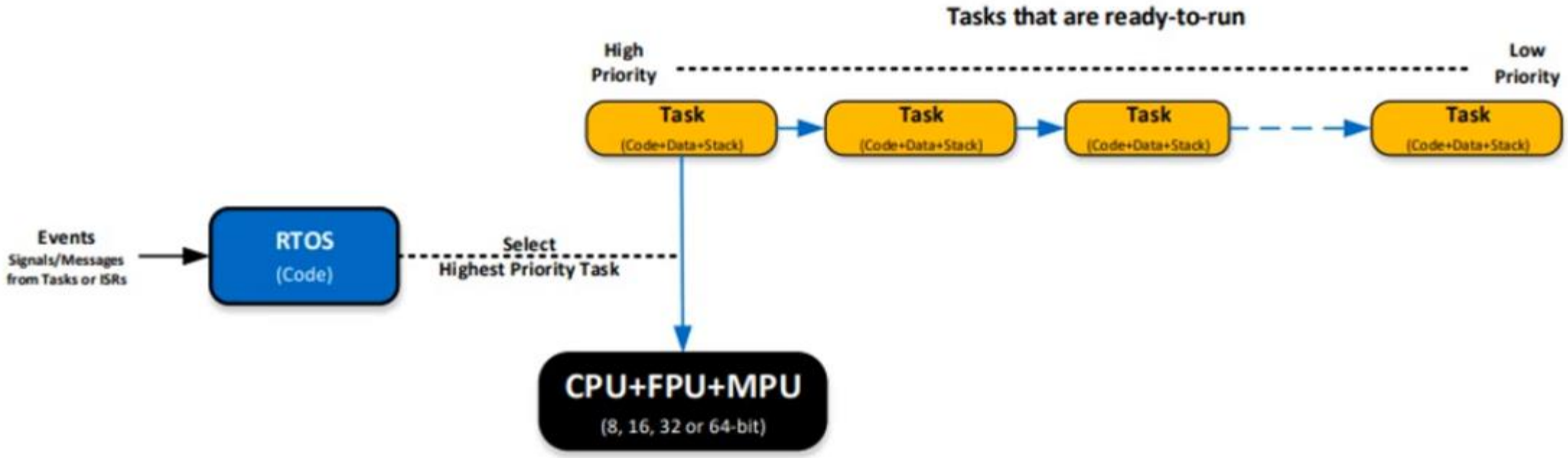

FreeRTOS
+-Source
has the core FreeRTOS kernel files
+-include
consists of The core FreeRTOS kernel header files
+-Portable
Consists of Processor specific code
+-Compiler x
contains all the ports created by compiler x
+-Compiler y
contains all the ports created by compiler y
+-MemMang
contains The sample heap implementations

FreeRTOS

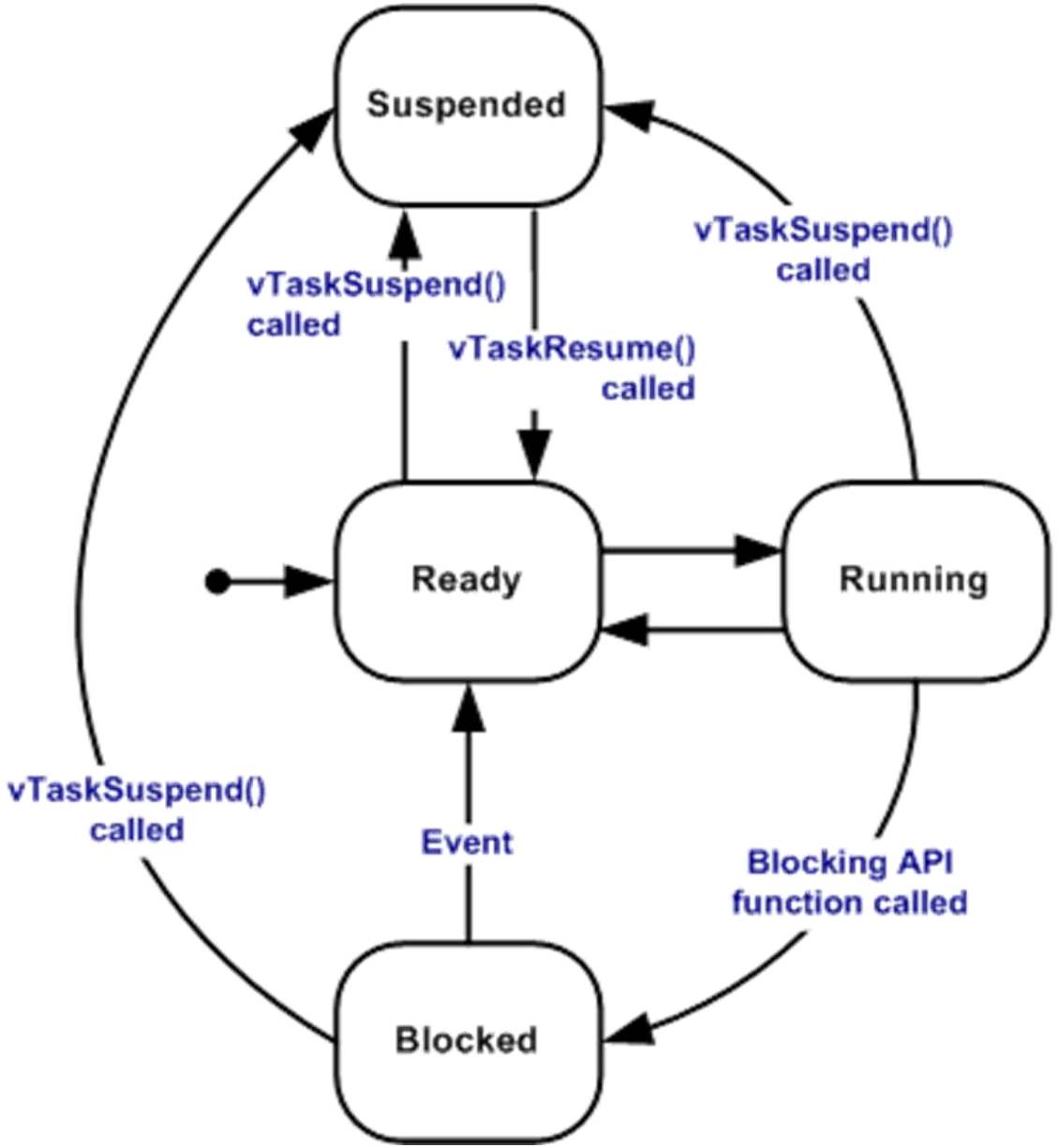

FreeRTOS.h
stddef.h
projdefs.h
FreeRTOSConfig.h
portable.h
portmacro.h
mpu_wrappers.h
FreeRTOS.h
list.c
croutine.c
timers.c
heap_3.c
queue.c
tasks.c

FreeRTOS Allow Multitasking

- An RTOS is a software that manages the time and resources of a CPU
  - Application is split into multiple tasks
  - The RTOS's job is to run the most important task that is *ready-to-run*
  - On a single CPU, only one task executes at any given time



Task States


Suspended
vTaskSuspend()
called
vTaskResume()
called
vTaskSuspend()
called
Ready
Running
vTaskSuspend()
called
Event
Blocking API
function called
Blocked

Task States

Running When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

Ready Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.4

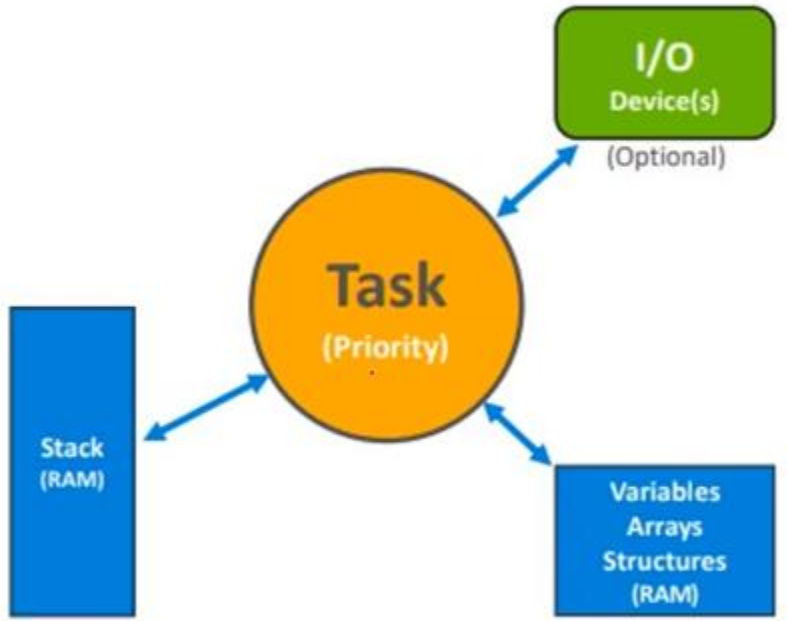
Blocked A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.
Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

Suspended Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

Tasks

- For each task:
  - YOU assign a priority based on its importance
  - Requires its own **Stack**
  - Manages its own variables, arrays and structures
  - Is typically an **infinite loop**
  - Possibly manages I/O devices
  - Contains YOUR application code

```
1   CPU_STK MyTaskStk[MY_TASK_STK_SIZE];  // Task Stack
2
3   void MyTask(void *p_arg)              // Task Code
4   {
5     Local Variables;
6
7     Task initialization;
8     while(1){
9       wait for Event;                 // Infinite Loop (Typ.)
10      Perform task operation;         // Do something useful
11
12    }
13  }
```



```
//Create the threads that will be managed by rtos
xTaskCreate(CloudUpdate,  "Cloud Update",     512, NULL, tskIDLE_PRIORITY + 1, &Handle_CloudUpdate);
Serial.println("Cloud Task Created");
xTaskCreate(LEDUpdate, "LED Update",          256, NULL, tskIDLE_PRIORITY + 2, &Handle_LEDUpdate )
Serial.println("LED Task Created");
xTaskCreate(BPUpdate,  "BP Update",           256, NULL, tskIDLE_PRIORITY + 3, &Handle_BPUpdate);
Serial.println("PB Task Created");
xTaskCreate(DACUpdate,  "DAC Update",         256, NULL, tskIDLE_PRIORITY + 1, &Handle_DACUpdate);
Serial.println("DAC Task Created");
}
```

Task Create

```
//Definition of tasks
static void CloudUpdate (void *pvParameters)
{
  SERIAL.println("Acquisition are Cloud Update:  Started");
  while(1)
  {
    SERIAL.print("\nCloud Update\n");
    SERIAL.flush();
    //ADC_Handler(ADC_ADC131);  //Acquire Signals
    ArduinoCloud.Update();
    msDelayMs(5000);
  }
  // delete ourselves.
  SERIAL.println("Cloud Update: Deleting");
  vTaskDelete(NULL);

}
```

Task 1

```
static void LEEDUpdate (void *pvParameters)
{
  SERIAL.println("LED Update:  Started");
  while(1)
  {
    LEDHandler();
    msDelayMs(100);
  }
  // delete ourselves.
  SERIAL.println("Cloud Update: Deleting");
  vTaskDelete(NULL);

}
```

Task 2

```
static void BPUpdate (void *pvParameters)
{
  SERIAL.println("Push Button Reading:  Started");
  while(1)
  {
    PBHandler();
    msDelayMs(100);
  }
}
```

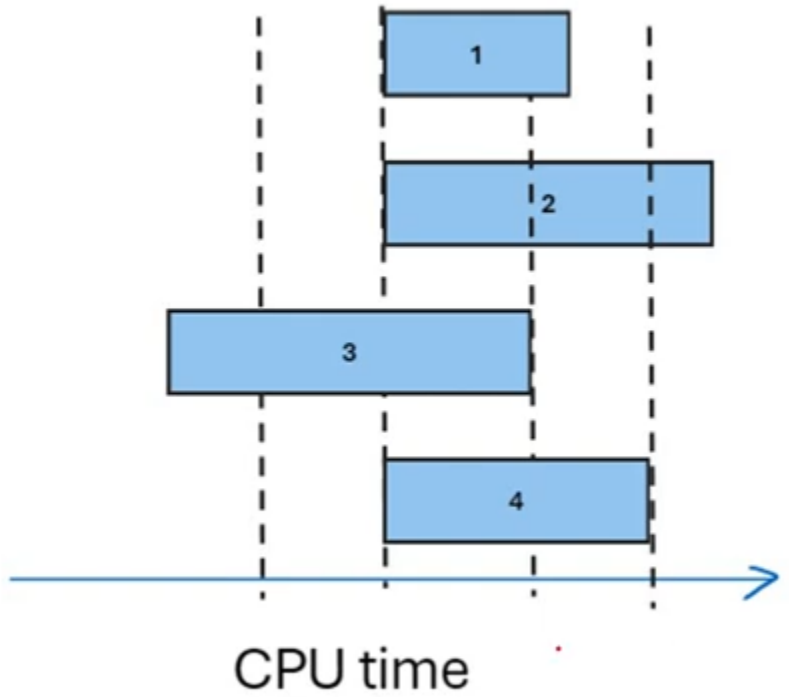
Task 3

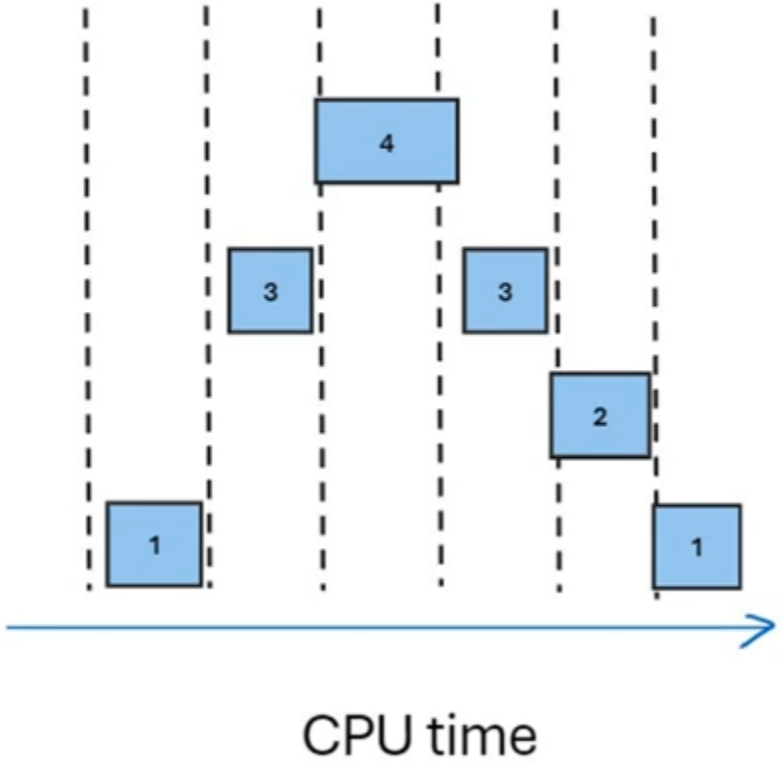
```
static void DACUpdate (void *pvParameters)
{
  SERIAL.println("DAC Update:  Started");
  while(1)
  {
    DACHandler();
    msDelayMs(1);
  }
}
```

Task 4

Task Define

Task taking CPU time

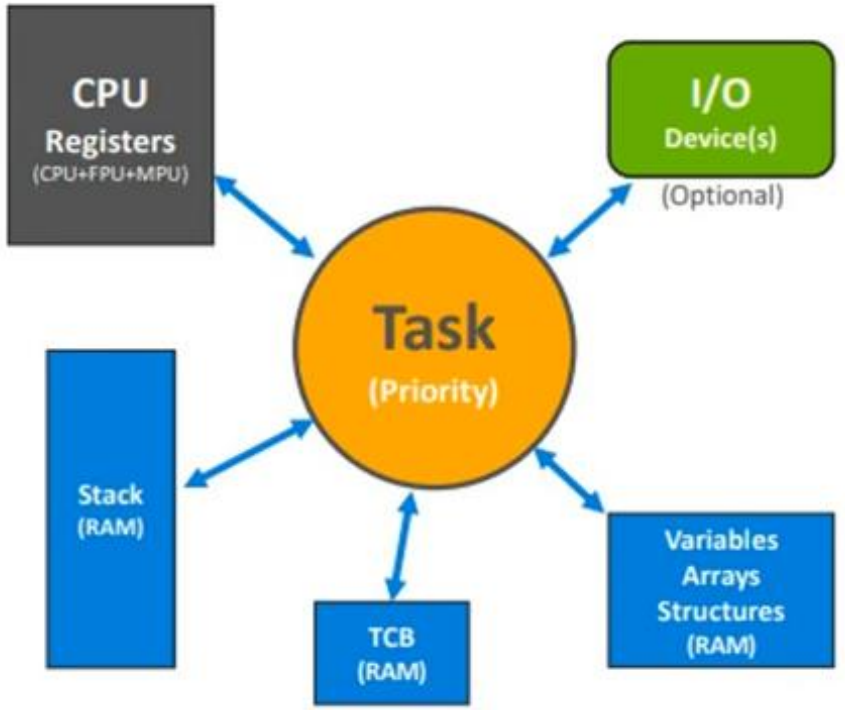




Creating Task

- YOU must tell the RTOS about the existence of a task:
  - The RTOS provides a special API: **OSTaskCreate()** or (equivalent)

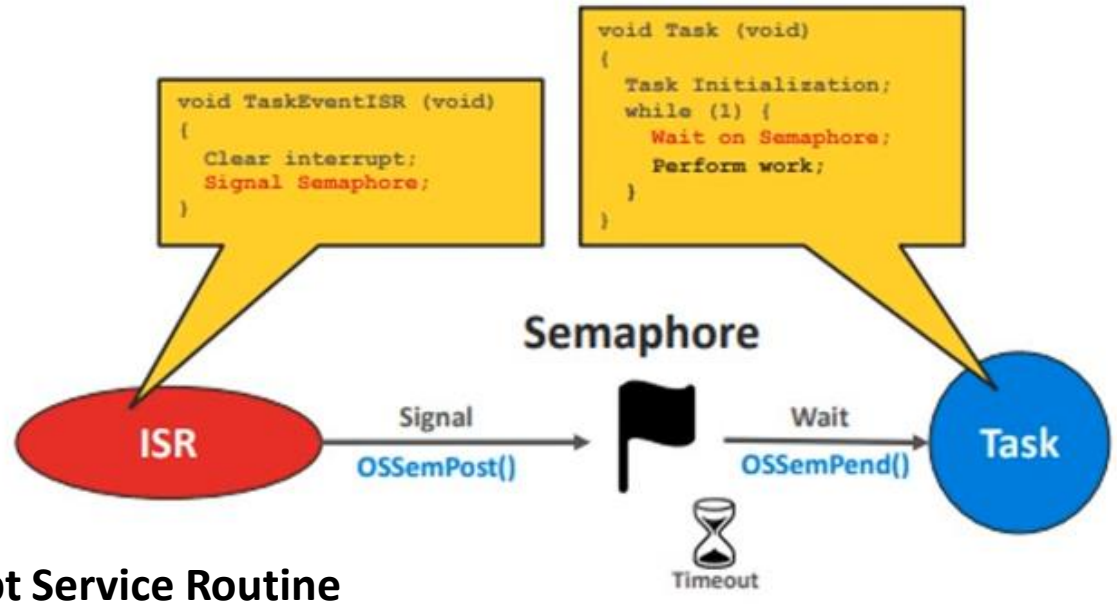
```
1  void OSTaskCreate(MyTask,                  // Address of code
2                    &MyTaskStk[0],          // Base of stack
3                    MY_TASK_STK_SIZE,       // Size of stack
4                    MY_TASK_PRIO,           // Task Priority
5                    :
6                    :);
```



- The RTOS assigns the task:
  - Its own set of **CPU registers**
  - A Task Control Block **(TCB)**

Signaling a task using Semaphore

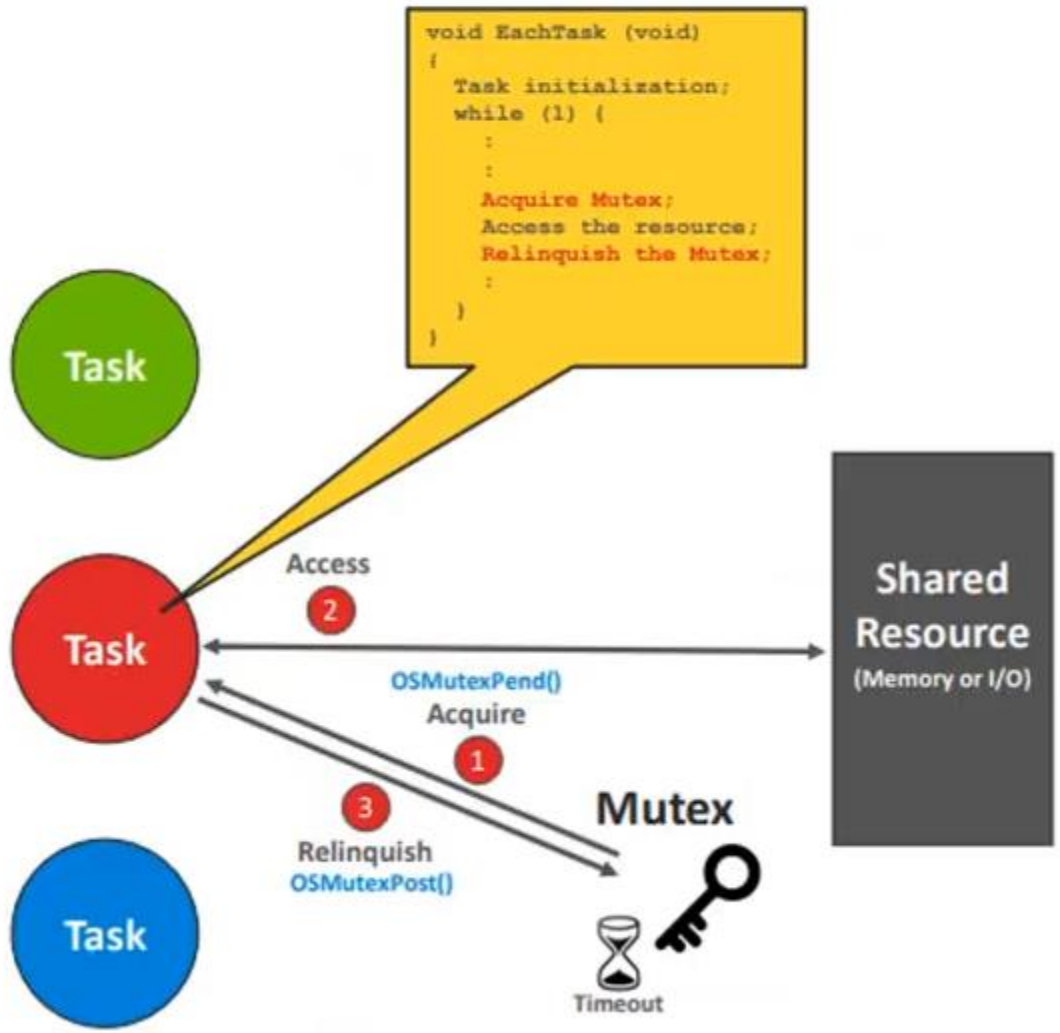
- Semaphore can be used to signal a task
  - Called from ISR or Task
  - Does not contain data



Interrupt Service Routine

Sharing a resource using mutex

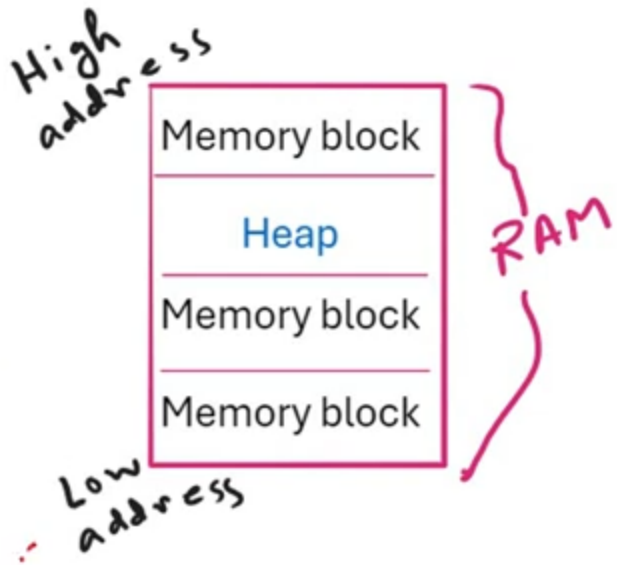
- What is a resource?
  - Shared memory, variables, arrays, structures
  - I/O devices
- RTOSs typically provide resource sharing APIs
  - Mutex have built-in **priority inheritance**
    - Eliminates unbounded priority inversions
  - There could be multiple mutexes in a system
    - Each protecting access to a different resource

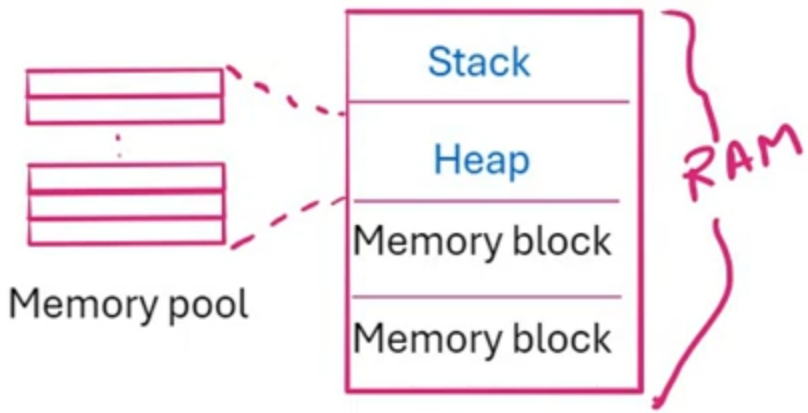


Memory allocation

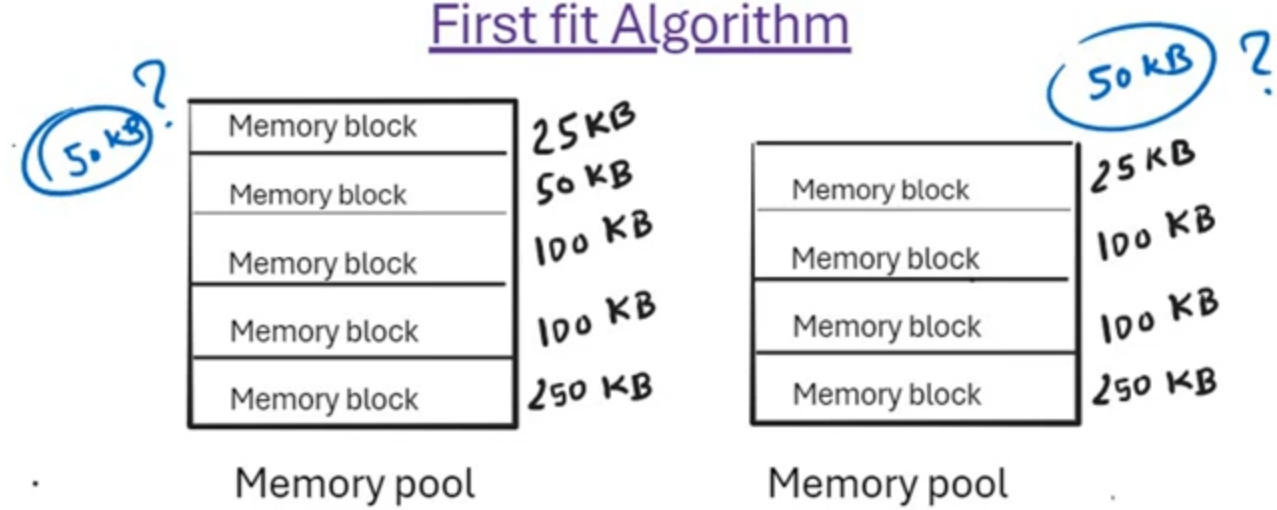
- Dynamic memory allocation
- Static memory allocation

Dynamic Allocation :
pvPortMalloc()
vPortFree()





First fit Algorithm



FreeRTOS files

- '**queue.c**' provide a way for tasks to send and receive messages or data packets to/from each other in a thread-safe manner.
- '**timers.c**' provides the functionality for creating and managing software timers.
- '**event groups.c**' used for inter-task communication and synchronization in a multitasking environment.
- '**croutine.c**', co-routines are a way to achieve cooperative multitasking within a single task context, allowing tasks to yield execution voluntarily to other tasks without relying on a preemptive scheduler.

List of Notation

- API Application Programming Interface
- CMSIS Cortex Microcontroller Software Interface Standard
- FIFO First In First Out
- HMI Human Machine Interface
- IDE Integrated Development Environment
- IRQ Interrupt Request
- ISR Interrupt Service Routine
- MCU Microcontroller
- MPU Memory Protection Unit
- RMS Rate Monotonic Scheduling
- RTOS Real-time Operating System
- SIL Safety Integrity Level
- TCB Task Control Block
- UART Universal Asynchronous Receiver/Transmitter