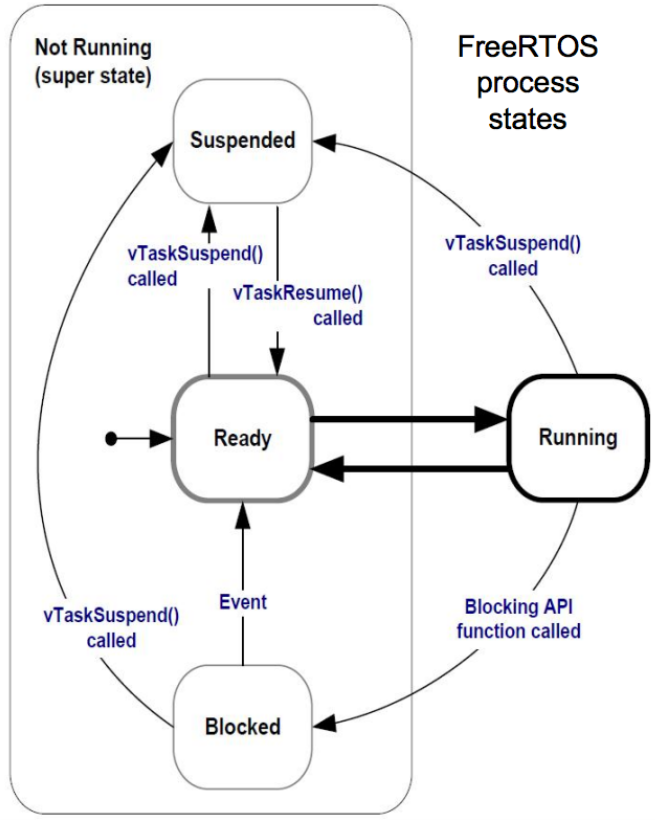
## 1**. Multitasking**

### **1.1.Process states**



Một process đang không ở trong running states thì sẽ ở trong các states sau:  
- Ready : sẵn sàng để thực thi nhưng chưa được thực thi do có các task khác với độ ưu tiên ngang bằng hoặc hơn đang chạy.  
- Blocked : process chưa sẵn sàng để chạy và phải chờ dữ liêu, ví dụ như data từ một queue khác  
- Suspended : Task không khả dụng để lên lịch (scheduling), process bị chuyển sang trạng thái hold và chỉ có thể được resume bởi OS call

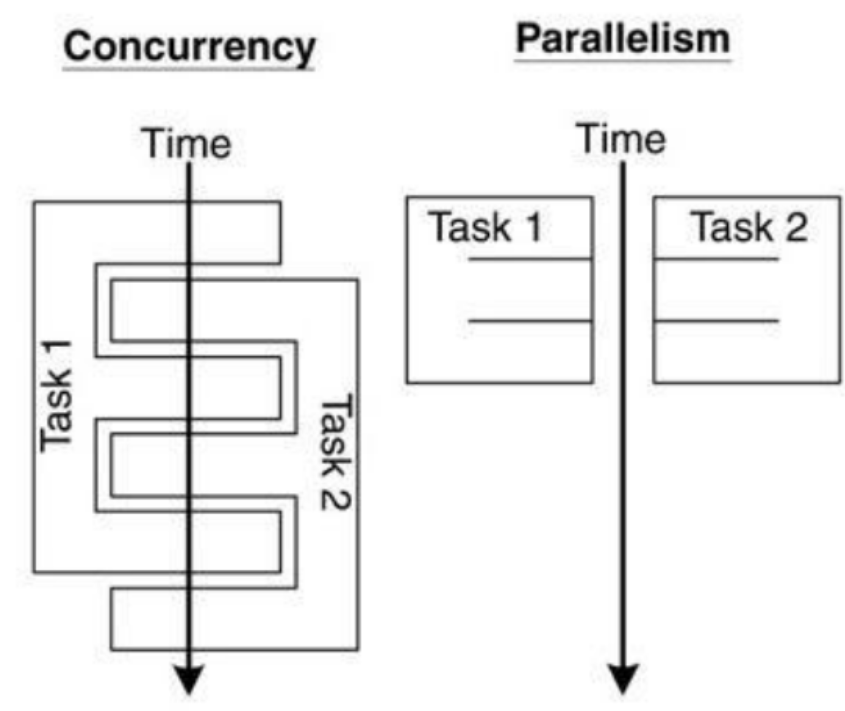
### **1.2. Multitasking types**

Có hai loại Multitasking  
- Pre-emptive: hệ thống có quyển chuyển 1 process từ trạng thái execute sang trạng thái chờ bất cứ khi nào và chuyển sang thực thi một process khác. Việc này thường được thực hiện thông qua interrupt even mặc dù process cũng có quyền yêu cầu.  
- Co-operative: process tự nguyện chuyển việc thực thi sang 1 process khác bằng cách gọi 1 function đặc biệt nào đó hoặc gọi bloking OS function.

### **1.3. Concurrency**

Concurrency: Multitasking in a single core environment

Một vi xử lý đơn nhân chỉ có thể thực thi duy nhất một process tại một thời điểm bất kì và các process khác phải nằm trong trạng thái non-running ( xem lại mục 2.1). Việc chuyển đổi từ một process này sang một process khác gọi là **context swtich**.



Ngày nay thì đa số là sử dụng vi xử lý đa nhân nên sẽ là **real parallelism** chứ không còn là **concurrency** so với vi xử lý đơn nhân nữa. Như đã đề cập ở fun fact trên, thời gian thực hiện context switch rất nhanh nên chúng ta có cảm giác là mọi thứ được thực hiện song song nhưng thực ra nó là one-by-one.

### **1.4. FreeRTOS schedule**

Trong FreeRTOS, mỗi một task sẽ được:

- Gán một priority

- Mỗi task chỉ có duy nhất một trạng thái vào bất kì thời điểm

- Chỉ duy nhất một task ở Running state vào bất kì thời điểm

Scheduler sẽ luôn chọn task đang ở Ready state và có priority cao nhất để đưa vào Running state.

### **1.5. Scheduling**

Để lập lịch cho Task, có 3 kỹ thuật được áp dụng:

* **Co-operative scheduling:** Mỗi task được thực thi đến khi kết thúc quá trình thì task tiếp theo mới được thực thi.
* **Round Robin Scheduling:** Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.
* **Preemptive Scheduling:** Phương pháp này ưu tiên phân bổ thời gian cho các task có mức ưu tiên cao hơn. Mỗi task được gán 1 mức ưu tiên duy nhất. Có thể có 256 mưc ưu tiên trong hệ thống, và có thể có nhiều task có cùng mức ưu tiên.

**Preemptive:** Các task có mức ưu tiên cao nhất luôn được kiểm soát bởi CPU, khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi, hoàn thành ISR sau đó hệ thống thực thi task có mức ưu tiên cao nhất tại thời điểm đó. Sau đó hệ thống mới tiến hành nối lại các task đang bị gián đoạn. Ở chế độ preemptive, hệ thống có thể đáp ứng các công việc khẩn cấp một cách nhanh chóng. Đa số các hệ thống thực tế đang chạy ở chế độ này.

**Non-preemptive:** Ở chế độ non-preemptive thì các task được chạy cho đến khi nó hoàn tất. Khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi và hoàn thành ISR , sau khi hoàn thành ISR thì hệ thống sẽ quay lại thực thi nốt phần việc còn lại của task bị gián đoạn. Task có mức ưu tiên cao hơn sẽ giành quyền kiểm soát CPU sau khi task bị gián đoạn thực thi xong. Ưu điểm của non-preemptive: độ trễ gián đoạn thấp. Nhược điểm của non-preemptive: do phải chờ task thực thi xong thì task có mức ưu tiên cao mới được thực thi, do đó mức đáp ứng của hệ thống thấp. Vì vậy rất ít hệ thống có sử dụng non-preemptive.

## **2. Cơ chế hoạt động của FreeRTOS**

Nhân (Kernel) của FreeRTOS hoạt động dựa vào 1 ngắt timer của chip, ngắt này tạo clock của RTOS gọi là Tick. Tùy từng chip mà Tick của FreeRTOS sẽ đặt ở các hàm ngắt khác nhau,

Ví dụ ở ARM-Cortex M thì Tick sẽ đặt trong hàm ngắt SysTick (timer hệ thống), đối với PIC32 thì Tick sẽ đặt trong hàm ngắt Timer 1, đối với AVR thì mình không rõ nhưng chắc chắn sẽ đặt trong 1 ngắt timer.

Chu kỳ ngắt thông thường sẽ là 1Khz, có thể thay đổi được tuy nhiên nếu thay đổi thì hàm vTaskDelay() sẽ chạy khác. Nếu đổi tần số Tick sang 10Khz thì muốn delay 10ms thì phải gọi vTaskDelay(100).

Mỗi lần Tick thì FreeRTOS sẽ thực hiện các hoạt động phân quyền chạy Task vụ. Tức là cho phép Task nào chạy, Task nào tạm dừng, Task nào chạy trước, Task nào chạy sau.

Đối với 1 chip có tốc độ chạy thấp ví dụ như PIC16, thì để đảm bảo sự realtime (RTOS chạy ổn) thì không nên có những ngắt liên tục với độ ưu tiên cao hơn ngắt Tick. Còn đối với chip tốc độ cao như STM32F4 thì có nhiều ngắt liên tục với ưu tiên cao hơn ngắt Tick thì không có vấn đề gì. Ở chip F4 mình đã từng thử 4 ngắt timer với tốc độ 200Khz có độ ưu tiên cao hơn ngắt Tick nhưng hệ thống vẫn hoạt động OK. Khi ngắt Tick xảy ra thì trong đấy sẽ gọi 1 hàm để hệ thống không thể xảy ra ngắt nữa gọi là hàm **enter\_critical\_section().** Tại sao vậy? Bởi vì khi Nhân RTOS đang hoạt động nó không muốn có bất kỳ ngắt nào xảy ra làm ảnh hưởng đến hoạt động của nó. Cho đến khi nhân hoạt động xong thì nó mới gọi hàm exit\_critical\_section() để cho phép các ngắt khác có thể xảy ra.

## **3. Interrupt management**

Trong Embedded System, khi có tín hiệu Interrupt, tác vụ đang chạy sẽ bị ngừng lại, thay vào đó **Interrupt Service Routine (ISR)** sẽ được thực thi. Sau khi ISR hoàn thành công việc, tác vụ ban đầu sẽ được thực thi tiếp.

Trong FreeRTOS, cần phân biệt sự khác nhau giữa Task và Interrupt:

- Task: là một “software feature” của FreeRTOS, không có sự khác biệt giữa các dòng MCU khác nhau mà FreeRTOS hỗ trợ. Task được quản lý bởi Scheduler (cũng là một thuật toán). Các thiết lập dành cho Task xảy ra ở tầng OS (cụ thể là FreeRTOS) bởi các API mà OS cung cấp;

- Interrupt Service Routine: là một “hardware feature” của MCU. Các thiết lập dành cho Interrupt xảy ra ở tầng hardware bằng cách thay đổi các giá trị trong thanh ghi (register) của MCU.

Interrupt luôn có độ ưu tiên cao hơn Task và không có cách nào để Task có thể chiếm quyền thực thi của ISR.

FreeRTOS có 2 hằng số quan trọng quy định độ ưu tiên của interrupt:

- configMAX\_SYSCAL\_INTERRUPT\_PRIORITY – độ ưu tiên tối đa của interrupt mà các interrupt-safe API có thể được sử dụng từ trong ISR;

- configKERNEL\_INTERRUPT\_PRIORITY – độ ưu tiên của RTOS kernel, luôn phải là độ ưu tiên thấp nhất trong các interrupt, giá trị này phụ thuộc vào từng dòng MCU.

### **3.1.Critical Section**

**Critical Section (hay Critical Region)** là vùng mà tất cả interrupt có mức độ ưu tiên thấp hơn hoặc bằng với **configMAX\_SYSCAL\_INTERRUPT\_PRIORITY** sẽ bị vô hiệu hóa. **Critical Section** được bao bọc bởi lệnh **taskENTER\_CRITICAL**() và **taskEXIT\_CRITICAL**(). Ở bên trong **Critical Section**, Scheduler sẽ ngừng hoạt động, do đó Context Switching sẽ không xảy ra. Đối với các interrupt bị vô hiệu hóa bên trong Critical Section, ISR của nó sẽ được thực thi ngay khi kết thúc Critical Section.

Những interrupt có độ ưu tiên cao hơn **configMAX\_SYSCAL\_INTERRUPT\_PRIORITY** không bị ảnh hưởng ở bên trong Critical Sections, nhưng không thể sử dụng các interrupt-safe API từ ISR của nó (với điều kiện macro **configASSERT**() đã được định nghĩa).

### **3.2. NVIC trong ARM**

Trong ARM có 1 bộ quản lý ngắt gọi là NVIC (Nested vectored interrupt controller). Bộ này cung cấp cho mỗi ngắt 1 thanh ghi chứa giá trị ưu tiên ngắt. Đây là thanh ghi 4bit tức là sẽ có tối đa 16 mức ưu tiên ngắt. Giá trị càng thấp thì mức ưu tiên càng cao, giá trị 0 sẽ là mức ưu tiên lớn nhất. 4bit của thanh ghi ưu tiên ngắt sẽ được chia ra làm 2 thành phần:

+ Ưu tiên ngắt chính (Preempt Priority)  
+ Ưu tiên ngắt phụ (Sub Priority)  
Chúng ta có thể cài đặt được Ưu tiên ngắt chính sẽ dùng bao nhiêu bits trong 4 bit của thanh ghi ưu tiên ngắt, bằng cách gọi hàm  
**NVIC\_PriorityGroupConfig(OPTION).** Có 5 OPTION như sau:  
+ NVIC\_PriorityGroup\_0: 0 bit preempt priority, 4 bit sub priority  
+ NVIC\_PriorityGroup\_1: 1 bit preempt priority, 3 bit sub priority  
+ NVIC\_PriorityGroup\_2: 2 bit preempt priority, 2 bit sub priority  
+ NVIC\_PriorityGroup\_3: 3 bit preempt priority, 1 bit sub priority  
+ NVIC\_PriorityGroup\_4: 4 bit preempt priority, 0 bit sub priority

Khi 2 ngắt xảy ra cùng lúc thì bộ NVIC sẽ dùng ưu tiên ngắt chính để xác định cái nào được ưu tiên chạy trước, nhưng nếu Ưu tiên ngắt chính như nhau thì NVIC sẽ duyệt đến Ưu tiên ngắt phụ để quyết định cái nào chạy trước

Trong FreeRTOS có 1 define để cài đặt sẽ sử dụng bao nhiêu bits để làm preempt priority. **#define configPRIO\_BITS**  
Do đó trong code tuyệt đối không gọi hàm **NVIC\_PriorityGroupConfig ()** khi sử dụng FreeRTOS, nếu có gọi thì gọi với OPTION tương ứng với define trên. Nếu không hệ thống của bạn sẽ treo đến vô tận luôn đó.

Khi chạy hàm **enter\_critical\_section()** thì trong hệ thống sẽ khổng thể có ngắt nào có thể xảy ra nữa, nhưng thực ra cũng tùy loại chip và tùy vào cài đặt của người dùng. Đối với ARM-CortexM3/4/7 thì 1 số ngắt vẫn có thể xảy ra ngay cả khi đã **enter\_critical\_section()** nếu cài đặt mức ưu tiên ngắt <<(dịch trái) 4 mà nhỏ hơn giá trị của thanh ghi **Base\_pri .** Vậy thì phải biết giá trị của thanh ghi Basepri bằng bao nhiêu chứ? Cách tính giá trị Base Pri như sau:  
**configMAX\_SYSCALL\_INTERRUPT\_PRIORITY =  
( configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY << (8 – configPRIO\_BITS) )**  
  
Trong đó:  
+ **configMAX\_SYSCALL\_INTERRUPT\_PRIORITY** : Chính là Base pri  
+ **configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY**: Giá trị priority lớn nhất mà các interrupt khác không nên vượt qua. Nếu đây là 1 thì các ngắt khác không nên đặt giá trị ưu tiên ngắt = 0. Trừ khi ngắt đó cần thiết có mức ưu tiên lớn hơn.  
+ **configPRIO\_BITS** : Là số bits dùng để mã hóa ưu tiên ngắt chính

Có 1 số lưu ý khi sử dụng ngắt ngoại vi như sau:  
+ Các ngắt có giá trị ưu tiên NVIC > BasePri mới có thể gọi được các hàm (API) của RTOS  
+ Không phải API nào của RTOS cũng được phép gọi trong interrupt, chỉ những API có đuôi FromISA mới được phép gọi trong interrupt. Bởi vì đây là các hàm an toàn.  
+ Tick RTOS không phải cái quan trọng nhất trong hệ thống, có 1 số cái quan trọng hơn, như USB, truyền thông, nhiều cái chỉ chậm 1 chút thôi sẽ bị hụt mất dữ liệu. Nên Tick RTOS không cần phải có mức ưu tiên cao nhất trừ khi ứng dụng của bạn không có truyền thông và cần đảm bảo độ real time.

### **3.3. Interrupt in FreeRTOS**

Nhưng có 1 sự khác nhau ở ngắt trong FreeRTOS và ngắt thông thường ở chỗ:  
Khi ngắt thông thường xảy ra nó sẽ nhảy vào hàm ngắt và các công việc cần phải làm đều đặt trong hàm ngắt. Còn ở FreeRTOS thì khi ngắt xảy ra nó sẽ nhảy vào hàm thủ tục ngắt, hàm thủ tục ngắt lại phát đi 1 "tín hiệu" (**give semaphore**) để hàm thực hiện chức năng của ngắt Unblocking (**take semaphore to unblock)** và hàm này có Priority cao hơn tất cả các task khác nên nó sẽ thực hiện ngay lập tức.

Tín hiệu ở đây chính là Semaphore, có 2 loại Semaphore là Binary Semaphore và Counting Semaphore.

#### 3.3.1. Binary Semephore

Đây giống như kiểu là 1 Queue nhưng mà chỉ có 1 phần tử nên trạng thái của nó chỉ có là Full hoặc Empty.  
Có 2 tác động chính vào Semaphore là "Take" và "Give", "Take" là dùng cho hàm Handler tức là hàm thực hiện chức năng của ngắt, khi mà hàm Handler gọi thủ tục này thì nó sẽ rơi vào trạng thái Block và chờ đợi sự kiện ngắt xảy ra. Tác động "Give" được thực hiện trong hàm thủ tục ngắt, nó sẽ phát ra tín hiệu là có ngắt xảy ra (Semaphore Full) và khi đó hàm đang đợi tín hiệu (cái mà đã gọi "Take" trước đó) sẽ ngay lập tức được Unblocking và thực hiện (Semaphore Empty). Sau khi thực hiện xong nó lại trở lại trạng thái block và chờ đợi cho sự kiện ngắt tiếp theo xảy ra.

##### 3.3.1.1. Binary Semephore API

**a. Khởi tạo semaphore**

**void vSemaphoreCreateBinary (xSemaphoreHandle xSemaphore);**          xSemaphore: là tên (handler) của semaphore.

**b. Hàm nhận semaphore “take” để unblocking**

**portBASE\_TYPE xSemaphoreTake ( xSemaphoreHandle xSemaphore, portTickType xTicksToWait );**  
 Hàm này được đặt trong Handler, và giá trị trả về của hàm là pdPASS hoặc pdFALSE.  
**xTicksToWait**: là thời gian đợi cho phép, thường là để thời gian đợi mãi mãi nên giá trị này sẽ là **portMAX\_DELAY**. Và nhớ trong file FreeRTOSConfig.h thì INCLUDE\_vTaskSuspend Set lên 1.  
**Lưu ý**: không dùng hàm này trong hàm ngắt.

**c. Hàm phát semaphore “give”**

**portBASE\_TYPE xSemaphoreGiveFromISR ( xSemaphoreHandle xSemaphore, portBASE\_TYPE \*pxHigherPriorityTaskWoken);**          Hàm này được đặt trong thủ tục ngắt, giá trị trả về là pdPASS hoặc pdFAIL.  
**pxHigherPriorityTaskWoken** là 1 cái giá trị đảm bảo là nó Semaphore sẽ Unblocking cái Task có Priority cao hơn cái đang chạy. Khi ở trong 1 hàm (ví dụ main chẳng hạn) phải định nghĩa biến này trước với kiểu là portBASE\_TYPE.  
         Nên set giá trị này là pdFALSE trước, sau đó gọi hàm "give", kiểm tra nó đã về pdTRUE chưa, nếu rồi thì thực hiện chuyển trạng thái?? Hic. cái này khó hiểu quá, có lẽ phải làm mới hiểu hết.

#### 3.3.2. Couting Semephore

Có 1 vấn đề xảy ra với Binary Semaphore là nếu mà tần suất ngắt xảy ra cao thì khi ngắt đầu tiên đang thực hiện, các ngắt tiếp theo xảy đến thì Semaphore cũng chỉ lưu lại được 1 tín hiệu do đó số lượng hàm phục vụ ngắt sẽ không đúng như mong muốn. Do đó Counting Semaphore ra đời để giải quyết vấn đề này, Semaphore này có thể chứa nhiều tín hiệu ngắt 1 lúc.

##### 3.3.2.1. Couting Semephore API

**a. Hàm khởi tạo**

**xSemaphoreHandle xSemaphoreCreateCounting ( unsigned portBASE\_TYPE uxMaxCount,  
                                                                                      unsigned portBASE\_TYPE uxInitialCount );**         Giá trị trả về của hàm là NULL (nếu khởi tạo không thành công, non-NULL nếu thành công )  
         **uxMaxCount:** số Semaphore tối đa có thể lưu.  
         **uxInitialCount:** số đếm khởi tạo.  
**ví dụ:**  
         xCountingSemaphore = xSemaphoreCreateCounting( 10, 0);

**Các hàm khác thì tương tự như Binary Semephore**

#### 3.3.3. Sử dụng Queue trong thủ tục ngắt

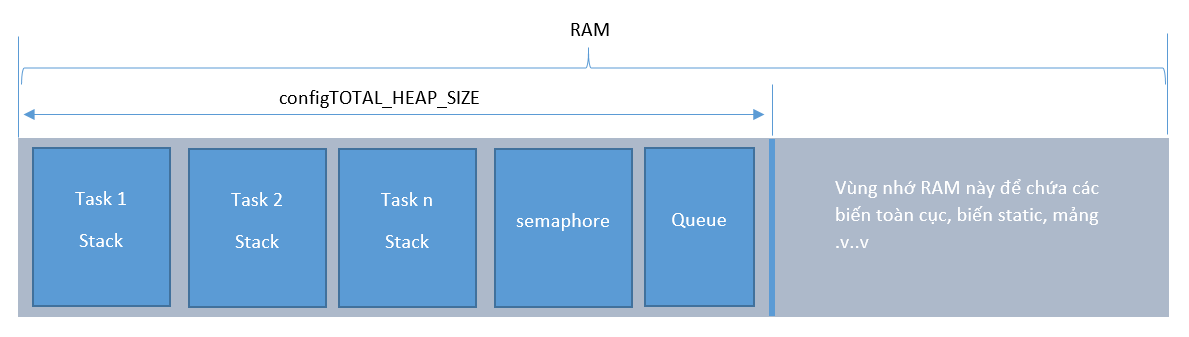
QueueSent là gửi và hàng đợi, QueueReceive là lấy dữ liệu từ hàng đợi.  
Vậy sự khác nhau khi sử dụng Counting Semaphore với Queue là gì? đó là  Semaphore thì dùng để kết nối với sự kiện, còn hàng đợi thì vừa kết nối sự kiện vừa truyền dữ liệu.  
Khi sử dụng hàng đợi ở trong ngắt thì phải sử dụng các hàm .....FromISR(). Cụ thể:  
**portBASE\_TYPE xQueueSendToFrontFromISR ( xQueueHandle xQueue, void \*pvItemToQueue,  
                                                                              portBASE\_TYPE \*pxHigherPriorityTaskWoken );  
portBASE\_TYPE xQueueSendToBackFromISR ( xQueueHandle xQueue, void \*pvItemToQueue,  
                                                                              portBASE\_TYPE \*pxHigherPriorityTaskWoken );**  
Giá trị trả về của các hàm này là pdPASS or pdQUEUE\_FULL;  
Cái **pvHigherPriorityTaskWoken** dùng giống như trong Semaphore.  
Người ta thường sử dụng cái này trong hàm UART để nhận và gửi tín hiệu ??  
**LƯU Ý**: Trong hàm thực hiện mà các công việc ngắt thì hàm nhận dữ liệu nên để thời gian chờ tối đa (không có time out), khi đó thì nếu mà không co dữ liệu thì hàm này sẽ bị Block.

#### 3.3.4. Interrupt Nesting

có 2 thông số cần chú ý ở file FreeRTOSConfig.h là  
configKERNEL\_INTERRUPT\_PRIORITY:   Giá trị Priority mà các hàm ngắt mang. (gt1)  
configMAX\_SYSCALL\_INTERRUPT\_PRIORITY:    Giá trị ưu tiên cao nhất mà hàm ngắt có thể có (gt2).  
giá trị của thông số này phải cao hơn giá trị của thông số phía trên.  
2 thông số này có ý nghĩa là những ngắt mà có ưu tiên bé hơn hoặc bằng (gt2) thì có thể bị cản trờ bởi 1 công việc đang khẩn cấp nào đó khác của Kernel, còn những ngắt có ưu tiên cao hơn sẽ được thực hiện 1 cách ngay lập tức và không bị vản trở bởi cái gì cả.  
  
LƯU Ý: Những ngắt mà không sử dụng các hàm API của Kernal thì có thể đặt bất cứ giá trị ưu tiên bất kỳ.  
Với ARM Cortex M3:  
Thì ngắt được định nghĩa ưu tiên dựa vào 1 con số ảo, con số này càng LỚN thì mức ưu tiên càng BÉ, có nghĩa là mức ưu tiên lớn nhất khi con số này là 0, và bé nhất là 255. (Hi. dịch thế thôi chứ đã là ARM bao giờ đâu mà biết :D )

## **4. FreeRTOS allocation memory**

FreeRTOS phân vùng nhớ RAM ra thành 2 vùng riêng biệt, vùng 1 gọi là vùng HEAP, vùng 2 là vùng còn lại của RAM,



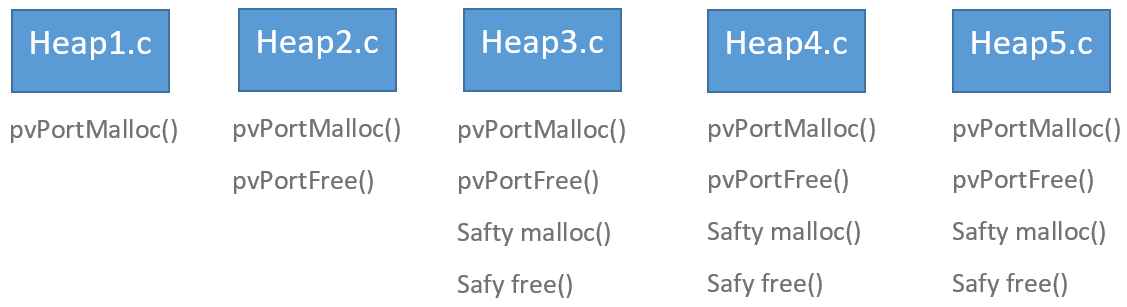
### FreeRTOS tạo ra 1 vùng nhớ gọi là vùng Heap, Mỗi Task được tạo ra sẽ chiếm lấy 1 vùng của vùng heap, vùng nhớ mà Task đó chiếm sẽ được Task đó tạo ra 1 vùng nhớ Stack. Đối với 1 ứng dụng thông thường không sử dụng RTOS thì stack và heap có ý nghĩa như sau: + Stack là vùng nhớ dùng để khai báo các biến, ví dụ: int bien = 0; float tong = 0; + Heap là vùng nhớ để cấp phát vùng nhớ động bằng lệnh malloc(). Tuy nhiên trong FreeRTOS các Task có thể tạo mới và xóa đi bất cứ lúc nào, nên được quản lý bởi vùng nhớ Heap, bản thân các Task có các vùng Stack riêng để khai báo biến. Vùng heap của FreeRTOS không chỉ dùng để chứa các Task mà còn chứa các biến được cấp phát động bằng hàm malloc() hoặc pvPortMalloc(), semaphore, Queue.

### Vùng nhớ còn lại của RAM dùng để chứa các biến toàn cục, biến static, mảng. Đồng nghĩa rằng các Task Static cùng được lưu ở phần này của RAM, chỉ các Dynamic Task mới được lưu trong vùng heap.

Dựa trên cấu tạo của vùng Heap thì ta có thể suy ra rằng, khi nào Task bị tràn stack thì có 1 phương pháp đơn giản là tăng configTOTAL\_HEAP\_SIZE lên , hoặc tăng kích thước stack cấp cho Task lúc CreateTask (xem phần 3)

### **4.1. Cấp phát động trong freertos**

Trong FreeRTOS nhà phát triển không khuyến khích sử dụng malloc() để cấp phát động mà họ đã tạo ra 1 hàm khác để cấp phát động riêng cho RTOS, đó là hàm pvPortMalloc(), chức năng của hàm pvPortMalloc() cũng y hệt như malloc(), nếu chạy hàm này mà trả về NULL thì tức là vùng nhớ heap có khả năng đã bị tràn. Tương ứng với hàm pvPortMalloc() thì sẽ có hàm pvPortFree() để thay thế hàm free() chuẩn của ngôn ngữ c. Lưu ý, đã sử dụng pvPortMalloc() thì không thể dùng free() để giải phóng bộ nhớ được và ngược lại. Để sử dụng được pvPortMalloc() và pvPortFree() thì cần phải sử dụng được đúng file heap\_x.c, các file heap hỗ trợ và không hỗ trợ như sau Hình 2.



+ Heap 1. Chỉ hỗ trợ cấp phát, không hỗ trợ giải phóng bộ nhớ. Nên nếu chỉ cần cấp pháp 1 lần hoặc không sử dụng cấp phát bộ nhớ động thì dùng heap1.c là được.  
+ Heap 2. Có hỗ trợ thêm giải phóng bộ nhớ, nhưng không hỗ trợ dồn các vùng nhớ vào gần nhau. Nên khi giải phóng xong, tổ chức vùng nhớ sẽ bị phân mảnh, do đó việc cấp phát các vùng nhớ lớn hơn sẽ không thể thực hiện được kể cả tổng vùng nhớ heap của hệ thống có đủ. **#define configAPPLICATION\_ALLOCATED\_HEAP** dùng để cài đặt địa chỉ đặt HEAP trong RAM.  
+ Heap 3. Hỗ trợ hàm malloc() và free() chuẩn của ngôn ngữ C, nhưng vẫn không nên dùng, lưu ý rằng cài đặt **configTOTAL\_HEAP\_SIZE** không có tác dụng ở heap3.  
+ Heap 4. Giống heap2, nhưng có thêm khả năng dồn các vùng nhớ phân mảnh thành 1 vùng nhớ lớn.  
+ Heap5. Gần giống heap4.c, nhưng các vùng nhớ trống không bị dồn lại với nhau mà sẽ được liên kết với nhau.

## **5. Các chức năng cơ bản của 1 RTOS**

* Bộ lập lịch (Scheduler).
* Các dịch vụ thời gian thực (Realtime Services).
* Đồng bộ và xử lý thông điệp (Synchronization and Messaging).

### 5.1. Scheduler

Mỗi task có thể có 3 trạng thái.

* **Ready to run:** Là trạng thái mà task đã có đủ các tài  nguyên để khởi chạy nhưng chưa chạy. Đây là trạng thái chuẩn bị của task.
* **Running:** Là trạng thái mà task đang được thực thi.
* **Blocked:** Khi task không có đủ các tài nguyên cần thiết để chạy thì nó sẽ được đưa về trạng thái blocked.

Để lập lịch cho Task, có 3 kỹ thuật được áp dụng:

* **Co-operative scheduling:** Mỗi task được thực thi đến khi kết thúc quá trình thì task tiếp theo mới được thực thi.
* **Round Robin Scheduling:** Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.
* **Preemptive Scheduling:** Phương pháp này ưu tiên phân bổ thời gian cho các task có mức ưu tiên cao hơn. Mỗi task được gán 1 mức ưu tiên duy nhất. Có thể có 256 mưc ưu tiên trong hệ thống, và có thể có nhiều task có cùng mức ưu tiên.

**Preemptive:** Các task có mức ưu tiên cao nhất luôn được kiểm soát bởi CPU, khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi, hoàn thành ISR sau đó hệ thống thực thi task có mức ưu tiên cao nhất tại thời điểm đó. Sau đó hệ thống mới tiến hành nối lại các task đang bị gián đoạn. Ở chế độ preemptive, hệ thống có thể đáp ứng các công việc khẩn cấp một cách nhanh chóng. Đa số các hệ thống thực tế đang chạy ở chế độ này.

**Non-preemptive:** Ở chế độ non-preemptive thì các task được chạy cho đến khi nó hoàn tất. Khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi và hoàn thành ISR , sau khi hoàn thành ISR thì hệ thống sẽ quay lại thực thi nốt phần việc còn lại của task bị gián đoạn. Task có mức ưu tiên cao hơn sẽ giành quyền kiểm soát CPU sau khi task bị gián đoạn thực thi xong. Ưu điểm của non-preemptive: độ trễ gián đoạn thấp. Nhược điểm của non-preemptive: do phải chờ task thực thi xong thì task có mức ưu tiên cao mới được thực thi, do đó mức đáp ứng của hệ thống thấp. Vì vậy rất ít hệ thống có sử dụng non-preemptive.

Kernel tiến hành quản lý task ở nhiều giai đoạn. Chúng bao gồm:

* Tạo task.
* Huỷ task.
* Thay đổi mức ưu tiên của task.
* Thay đổi trạng thái của task.

### 5.2. RTOS services

Kernel là trái tim của mỗi hệ điều hành. Nó thực hiện chức năng quản lý và lập lịch các quá trình sử dụng CPU và điều phối tài nguyên. Mỗi task chỉ được thưc thi bởi CPU trong một khoảng thời gian, trong các khoảng thời gian còn lại thì task được quản lý bởi các dịch vụ quản lý của hệ điều hành.

Các dịch vụ của RTOS bao gồm:

* Xử lý ngắt (Interrupt handling services).
* Dịch vụ quản lý thời gian (Time services).
* Dịch vụ quản lý thiết bị (Device management services).
* Dịch vụ quản lý bộ nhớ (Memory management services).
* Dịch vụ quản lý các kết nối Vào - Ra (IO services).

### 5.3. Message

Các thông điệp sử dụng để trao đổi thông tin giữa các hệ thống khác nhau hoặc giữa các task. Các dịch vụ quản lý thông điệp bao gồm:

* Semaphores
* Event flags
* Mailboxes
* Pipes
* Message queues

**Semaphores**: Dùng để đồng bộ hóa quyền truy cập vào các tài nguyên dùng chung.

**Event Flags**: Dùng để đồng bộ hóa các hoạt động cần có sự phối hợp của nhiều task.

**Mailboxes, Pipes, Message queues**: Dùng để quản lý các thông điệp gửi đi - đến giữa các task.