# Lesson 1: Introduction RTOS

## BLOCKING

Blocking và Non-Blocking trong lập trình chủ yếu được đề cập khi muốn miêu tả về cách một chương trình thực hiện các dòng lệnh của nó. Chúng ta có thể hiển một cách đơn giản, nếu chương trình được thực hiện theo mô hình Blocking có nghĩa là các dòng lệnh đucợ thực hiện một cách tuần tự. Khi một dòng lệnh ở phía trước chưa hoàn thành thì các dòng lệnh phía sau sẽ chưa được thực hiện và đợi khi mà thao tác phía trước hoàn tất, và nếu như các dòng lệnh trước là các thao tác cần nhiều thời gian xử lý như liên quan đến IO (input/output) hay mạng (networking) thì bản thân nó sẽ trở thành vật cản trở (blocker) cho các lệnh xử lý phía sau mặc dù theo logic thì có những việc ở phía sau ta có thể xử lý được luôn mà không cần phải đợi vì chúng không liên quan gì đến nhau.

Mô hình blocking tồn tại từ lịch sử, khi mà máy tính chỉ có thể xử lý đơn nhiệm trên một lõi (core) của bộ vi xử lý (chip). Nhưng theo thời gian, công nghệ ngày một trưởng thành với những thành tựu về phần cứng, máy tính giờ có thể làm nhiều việc cùng một lúc thì người ta cần phải suy nghĩ đến việc làm sao tận dụng được tối đa tài nguyên xử lý của máy tính và tránh lãng phí nó. Từ đó mầ bất cứ chỗ nào có phần xử lý Blocking không cần thiết, người ta cần thay vào một giải pháp sử dụng tài nguyên khôn ngoan hơn, đó là Non-Blocking.

Trong mô hình Non-Blocking, các dòng lệnh không nhất thiết phải lúc nào cũng phải thực hiện một cách tuần tự (squential) và đồng bộ (synchronous) với nhau. Ở mô hình này nếu như về mặt logic dòng lệnh phía sau không phụ thuộc vào kết quả của dòng lệnh phía trước, thì nó cũng có thể hoàn thành được thực hiện ngay sau khi dòng lệnh phía trước được gọi mà không cần đợi cho tới khi kết quả được sinh ra. Những dòng lệnh phía trước miêu tả ở trên còn có thể gọi là được thực hiện theo cách không đồng bộ (Asynchronous), và đi theo mỗi dòng lệnh thường có một callback (lời gọi lại) là đoạn mã sẽ thực hiện ngay sau khi có kết quả trả về từ dòng lệnh không đồng bộ. Để thực hiện mô hình Non-Blocking, người ta có những cách để thực hiện khác nhau, nhưng về cơ bản vẫn dựa vào việc dùng nhiều Thread (luồng) khác nhau trong cùng một Process (tiến trình), hay thậm chí nhiều Process khác nhau (inter-process communication - IPC) để thực hiện. Và mẫu thiết kế (design pattern) tên là event-loop là một trong những mẫu thiết kế nổi tiếng để thực hiện cơ chế Non-Blocking.

VD: Blocking and Non-Blocking

Phần phía trên miêu tả sự hoạt động theo cơ chế Blocking mà ở đây mặc dù không có sự liên đới giữa 3 việc, nhưng các công việc tiếp sau luôn phải chờ công việc phía trước thực sự xong rồi mới có thể bắt đầu thực hiện. Các bước sẽ được mô tả như dưới đây

* Hàm dataSync1.get() được gọi để lấy dữ liệu, vì nó là Blocking nên trước khi công việc này hoàn thành các việc tiếp sau sẽ phải đợi
* Hàm printData(d1) được gọi để in dữ liệu lấy về từ dataSync1.get(), tương tự nó cũng là Blocking
* Hàm dataSync2.get() được gọi để lấy dữ liệu, mặc dùng là nó không liên quan gì đến hai dòng lệnh trên, nhưng đến tận bây giờ nó mới được thực hiện và là Blocking nên chiếm một khoảng thời gian xử lý nữa
* Hàm printData(d2) được gọi để in dữ liệu lấy về từ dataSync2.get(), là Blocking
* Hàm dataSync3.get() được gọi để lấy dữ liệu, là Blocking
* Hàm printData(d3) được gọi để in dữ liệu lấy về dataSync3.get(), là Blocking

=> Ở phần này, mọi thao tác đều là blocking nên thời gian để thực hiện xong hết các thao tác sẽ bằng tổng thời gian của từng thao tác.

Phía dưới là phần thể hiện việc làm tất cả những việc trên, các thao tác in dự liệu printData(d1), printData(d2), printData(d3) vẫn là các thao tác Blocking nhưng ở đây có sự tham gia của Non-Blocking trong các thao tác lấy dữ liệu dataAsync1.get(), dataAsync2.get(), dataAsync3.get(). Các thao tác Non-Blocking sẽ được bắt đầu gần như ngay lập tức và không cần phải chờ các thao tác phía trước thực hiện xong. Sau khi có kết quả các thao tác Non-Blocking sẽ gọi lại callback để in kết quả trả về ra màn hình. Cụ thể sẽ được diễn giải như ở dưới đây:

* Hàm dataAsync1.get() được gọi để lấy dự liệu, vì nó là Non-Blocking nên quá trình thực thi sẽ không phải dừng ở đây mà tiếp tục thực hiện dòng lệnh tiếp sau gần như ngay lập tức, tất nhiên vẫn phải sau khi đăng ký một callback để in ra dữ liệu trả về từ dataAsync1.get().
* Như nói ở trên, ngay sau đó, hàm dataAsync2.get() được gọi cùng với đăng ký callback. Vì là Non-Blocking nên quá trình cũng giống như trên.
* Tiếp theo hàm dataAsync3.get() cũng được thực hiện tương tự. Đến đây, 3 hàm gọi để lấy dữ liệu gần như được thực hiện đồng thời mà không cần phải chờ nhau.
* Trong khi hàm dataAsync1.get() và dataAsync3.get() đang thực hiện thì hàm dataAsync2.get() đã lấy được dữ liệu về, lúc này callback được gọi để in dữ liệu đó ra màn hình, trong callback lúc này printData(d2) được gọi và nó là Blocking.
* Trong thời gian printData(d2) đang thực hiện, dataAsync1.get() đã hoàn tất việc lấy dữ liệu, callback của nó được gọi tuy nhiên vì printData(d2) là Blocking và đang thực hiện nên việc thực hiện printData(d1) sẽ phải chờ.
* Cũng tương tự như trên, dataAsync3.get() cũng hoàn tất việc lấy dữ liệu, callback của nó được gọi, lần này printData(d3) không những phải chờ printData(d2) như trên mà nó còn phải chờ thêm cả printData(d1) bởi vì printData(d1) cũng là Blocking. Sau khi cả printData(d2) và printData(d1) được hoàn thành thì printData(d3) được thực hiện và toàn bộ quá trình hoàn tất.

=>Bây giờ nhìn lại hình vẽ một lần nữa ta có thể thấy Non-Blocking rút ngắn thời gian thực hiện chương trình hơn là Blocking, việc rút ngắn thời gian này không phải vì các công việc được thực hiện nhanh hơn mà vì nhiều việc được thực hiện cùng một lúc hơn.

## Task

Task: 1 công việc cụ thể mà vi điều khiển cần phải hoàn thành.

( Một task là một chương trình, chương trình nầy chạy liên tục trong vòng lặp vô tận và không bao giờ dừng lại. Với mỗi task thì có niềm tin duy nhất là chỉ mình nó đang chạy và có thế sử dụng hết nguồn tài nguyên sẵn có của bộ xử lý (mặc dù là thực tế thì nó vẫn phải chia sẽ nguồn tìa nguyên này với các task khác).)

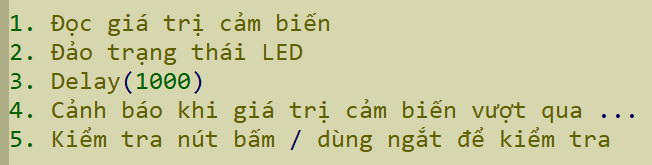
Lấy một ví dụ đơn giản như sau. Một vi điều khiển cần làm những công việc sau:

Task 1: Blink LED với chu kỳ 1s.

Task 2: Đọc cảm biến khí gas để cảnh báo nguy hiểm khi cần.

Task 3: Đọc nút bấm để điều chỉnh chu kỳ nháy led.

Vậy mình sẽ viết chương trình theo thứ tự như sau:



Nhìn vào cách làm trên thì việc đọc cảm biến và cảnh báo có thể bị lỡ, do việc đọc diễn ra 1 lần/ 1s. Có thể dẫn đến nguy hiểm. Ở đây mình mới chỉ sử dụng 3 task. Trong hệ thống lớn số lượng task sẽ lớn hơn rất nhiều, và lượng task cần đảm bảo hoạt động đúng thời cho phép (ảnh hưởng đến an toàn người dùng) là rất nhiều.

Lập trình vi điều khiển theo cách polling + interrupt thông thường đang không đáp ứng được những yêu cầu đề ra, dẫn đến cần một cơ chế để giúp nó thực hiện các task đúng thời gian người dùng quy định. Từ đó, người ta sử dụng RTOS.

## RTOS

### OS (Operating System)

Một thiết bị sẽ có nhiều tác vụ chạy cùng lúc, cần một phần mềm để quản lý tài nguyên chung giữ các tác vụ, điều khiển các ngoại vi.

OS là một phần mềm quản lý tài nguyên và sắp xếp lịch cho các tác vụ trong hệ thống.

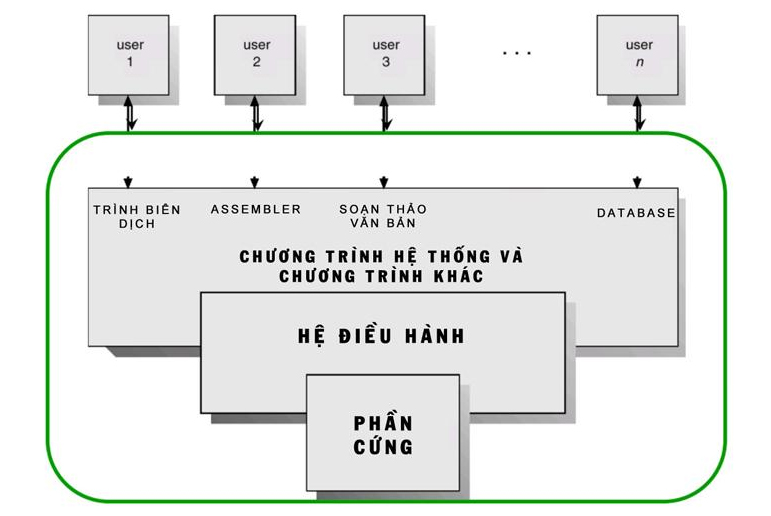
OS là một chương trình quản lý tất cả những yếu tố như phần cứng và phần mềm của máy tính.

Trong đó:

* Phần cứng gồm: CPU, RAM, ROM, thiết bị nhập (chuột, bàn phím…), cùng với các thiết bị xuất (loa, máy in…)
* Phần mềm gồm: có chương trình hoặc các ứng dụng có trong máy tính.

**Thành phần chính của OS:**

* Kernel: cung cấp những điều khiển dừng lại ở mức độ cơ bản đối với các thiết bị phần cứng của máy tính. Vai trò chính của chúng bao gồm: Đọc và ghi dữu liệu vào bộ nhớ, tiến hành xử lý những lệnh thực hiện và xác định cách mà dữ liệu được nhận cũng như được gửi bởi các thiết bị khác.
* API - giao diện lập trình ứng dụng: API sẽ cho phép những nhà phát triển ứng dụng có thể viết các mã module.
* Giao diện người dụng: User Interface sẽ cho phép hệ thống tương tác với người dùng thông qua những graphical icons và 1 desktop hoặc có thể thông qua một command line.



**Hệ điều hành OS đang được chia thành các loại như sau:**

* Multi-user (nhiều người dùng): Hệ thống cho phép nhiều người dùng có thể chạy các chương trình trong cùng một thời điểm. Một số hệ điều hành lớn hơn sẽ cho phép số lượng người dùng đến hàng trăm và thậm chí là hàng ngàn người cùng dùng đồng thời.
* Multitasking (đa nhiệm) sẽ cho phép nhiều chương trình có thể chạy đồng thời một lúc.
* Mutilthread (đa luồng): hệ thống cho phép những phần khác nhau của một chương trình có thể chạy cùng lúc.
* Real time (thời gian thực): hệ thống sẽ phản hồi ngay lập tức.

**Chức năng chính của hệ điều hành OS:**

* **Quản lý chia sẽ tài nguyên:** Trên một hệ thống máy tính sẽ sở hữu rất nhiều tài nguyên như CPU, bộ nhớ, các thiết bị ngoại vi,... khá giới hạn. Tuy nhiên, ở trong hệ thống đa nhiệm có nhiều yêu cầu về việc cung cấp tài nguyên từ nhiều chương trình. Chính vì vậy, hệ điều hành cần phải phân phối một cách nhanh chóng các tài nguyên hiện đang có sẵn.
* **Kiểm soát chương trình:** Hệ điều hành OS sẽ giúp kiểm soát những chương trình hiện đang chạy ở trên hệ thống. Mục đích chính là để tránh xảy ra các lỗi cũng như trường hợp mà người dùng sử dụng không đúng cách. **(?)**

Một số chức năng khác có thể kể đến như:

* Process Management - Quản lý quá trình
* Memory Management - Quản lý bộ nhớ
* Storage Management - Quản lý ổ cứng
* USB, HDD - Quản lý bộ nhớ ngoài
* Quản lý hệ thống lưu trữ
* Giúp người sử dụng có thể tiếp cận một cách nhanh chóng và tiện lợi.

### RTOS (Real-time OS)

RTOS là hệ điều hành thời gian thực sử dụng trong những ứng dụng yêu cầu thời gian đáp ứng nhanh, chính xác về thời gian. Thời gian thực ở đây có thể bị hiểu nhầm là thực thi công việc rất nhanh. Nhưng thật ra ở đây nó mang ý nghĩa là thời gian đáp ứng nhanh, đúng như người dùng mong muốn.

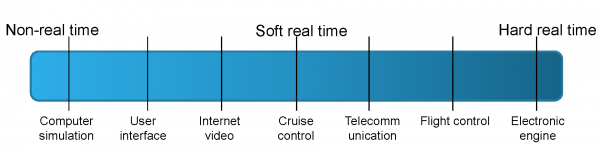
Ví dụ: Người dùng muốn hệ thống thực hiện nếu đo được khí gas bị rò thì sẽ gửi cảnh báo sau không quá 0.5s thì hệ thống đáp ứng sau 0.1s hay 0.4s đều là real-time cả. Trong khi ngay cả nếu người dùng muốn cảnh báo sau không quá 10s, thì việc hệ thống đáp ứng sau 9s, mặc dù con số này khá lớn (coi như là khá chậm) thì vẫn là đáp ứng thời gian thực real-time.

**So sánh với hệ điều hành thông thường :**

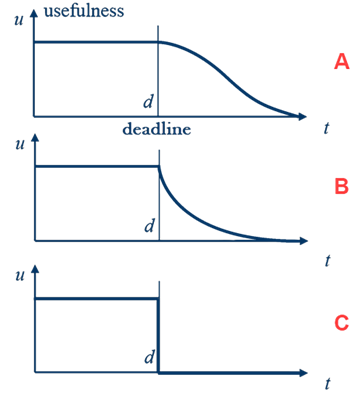
* RTOS khác với các hệ điều hành thông thường trong máy tính như window hay linux. Với hệ điều hành thông thường, khi bạn mở ứng dụng lên thì sẽ có nhiều lúc bạn phải chờ khá lâu, hoặc đôi khi ứng dụng lỗi thì chúng ta chỉ cần đóng process rồi chạy lại. Việc chờ đợi như vậy hầu như không ảnh hưởng gì nhiều lắm, chỉ làm người dùng hơi khó chịu thôi.
* Trong khi đó, RTOS được thiết kế ra cho các nhiệm vụ đặc biệt. Các ứng dụng cần được thực thi với thời gian thật chính xác đặt trước, các lỗi phát sinh cần được cô lập và xử lý nhanh chóng. Mọi sự chậm trễ, lỗi phát sinh không lường trước có thể gây lỗi hệ thống, nặng hơn là gây hậu quả nghiêm trọng.
* Với bộ tài nguyên khá nhỏ của mình, các hệ nhúng (vi điều khiển) sẽ cố gắng thực hiện một số chức năng chính: tối ưu tối đa số luồng, bộ lập lịch và các tác vụ (task). Với việc chỉ có một core, nên việc chạy song song các task là không thể. RTOS sẽ có tính năng lập lịch, để việc thực thi các task gần như là song song, và các task sẽ không cần phải đợi quá lâu để được thực thi.

**Hệ điều hành RTOS thường được chia thành ba loại chính là:**

* Hard RTOS (Tạm dịch: Hệ điều hành thời gian thực cứng): Các hệ điều hành này sẽ luôn đảm bảo các tác vụ được hoàn thành trong một khoảng thời gian cố định, xác định cụ thể, không có sự sai sót.
* Soft RTOS (Tạm dịch: Hệ điều hành thời gian thực mềm) : Đây là các hệ điều hành với thời gian thực hiện tác vụ có thể nhận được một số nới lỏng trong phạm vi cho phép, chỉ cần hoàn thành đúng thời gian quy định.
* Firm RTOS (Tạm dịch: Hệ điều hành thời gian thực bền vững) hay còn gọi là Non Real-time: Hệ điều hành cũng có các giới hạn thời gian được xác định cụ thể với mức độ chính xác cao nhất, Firm RTOS cũng đảm bảo các tác vụ luôn được thực hiện thành công ngay cả trong trường hợp quá thời hạn quy định.



Với mô phỏng máy tính sẽ là non-real time ở cấp độ cao nhất, tiếp đến là giao tiếp người dùng, truyền video thông qua internet, tới mức soft real time là hệ thống cruise control trên ô tô (hệ thống tự kiểm soát tốc độ), các ứng dụng về viễn thông, điều khiển trên máy bay,động cơ điện thì yêu cầu mức hard real time, bạn thử nghĩ xem nếu như đi máy bay mà máy bay hạ xuống đường băng rồi mà 1,2s sau mới có thông báo thì sẽ như thế nào ?



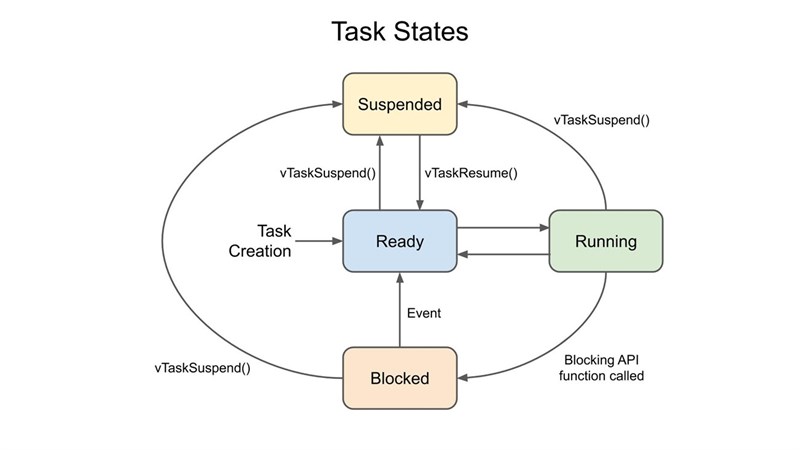
A: non-realtime, B: soft-realtime, C: hard-realtime  
​

* Hard-realtime: được thiết kế để hoạt động trong các giới hạn thời gian rất nghiêm ngặt, trong đó việc bỏ lỡ thời hạn (deadline) có thể dẫn đến hậu quả thảm khốc. Ví dụ về các hệ thống thời gian thực cứng bao gồm hệ thống kiểm soát không lưu, hệ thống điều khiển cho các nhà máy điện hạt nhân. Các hệ thống này phải đáp ứng thời hạn một cách chắc chắn tuyệt đối và không thể chấp nhận bất kỳ sự chậm trễ hoặc lỗi hệ thống nào.
* Soft-realtime: được thiết kế để hoạt động trong các ràng buộc thời gian linh hoạt hơn. Các hệ thống này có thể có thời hạn quan trọng, nhưng việc bỏ lỡ thời hạn không dẫn đến hậu quả thảm khốc. Ví dụ về các hệ thống thời gian thực mềm bao gồm các ứng dụng đa phương tiện như truyền phát video, gaming trực tiếp. Mặc dù các hệ thống này yêu cầu xử lý kịp thời, nhưng chúng có thể chịu được một số độ trễ mà không ảnh hưởng đến nhiều trải nghiệm chung của người dùng.
* Non-realtime: hệ thống trong đó thời gian phản hồi không quan trọng và không có các ràng buộc nghiêm ngặt về thời gian xử lý. Thời gian phản hồi có thể khác nhau tùy thuộc vào tải trên hệ thống và tài nguyên hệ thống.

**Các chức năng cơ bản của RTOS**

Scheduler (Bộ lập lịch), trong Scheduler mỗi tác vụ sẽ có 3 trạng thái mặc định là:

* Ready to run: Trạng thái chuẩn bị của tác vụ khi đã có đủ các tài nguyên để khởi chạy.
* Running: Trạng thái tác vụ đang thực thi.
* Blocked: Đây là trạng thái mà các tác vụ không đủ tài nguyên để xử lý sẽ được về trạng thái khóa.



**RTOS Services (Dịch vụ thời gian thực) với các dịch vụ:**

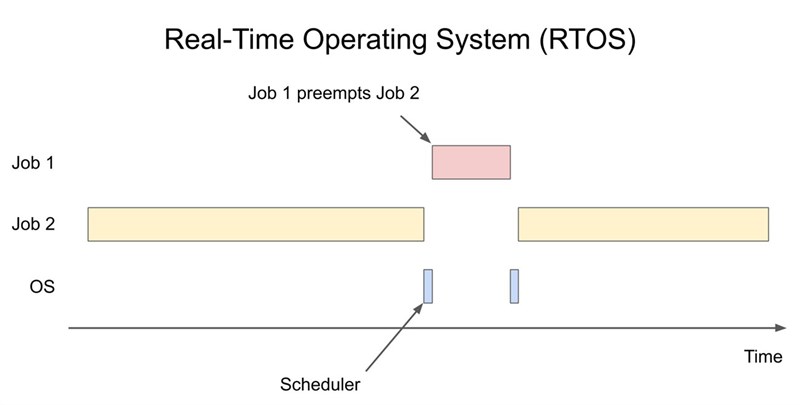
* Dịch vụ xử lý ngắt (Interrupt handling services).
* Dịch vụ thời gian (Time services).
* Dịch vụ quản lý thiết bị (Device management services).
* Dịch vụ quản lý bộ nhớ (Memory management services).
* Dịch vụ quản lý kết nối (IO services).

**Mesaging (Các thông điệp). Các thông điệp này sẽ dùng để trao đổi thông tin giữa các tác vụ với nhau, bao gồm:**

* Semaphores: Đồng bộ hóa quyền truy cập các tài nguyên dùng chung.
* Event flags: Đồng bộ hóa hoạt động cần có sự phối hợp của nhiều tác vụ.
* Mailboxes, Pipes, Message queues: Quản lý các thông điệp đã được gửi đi và đến giữa các task.

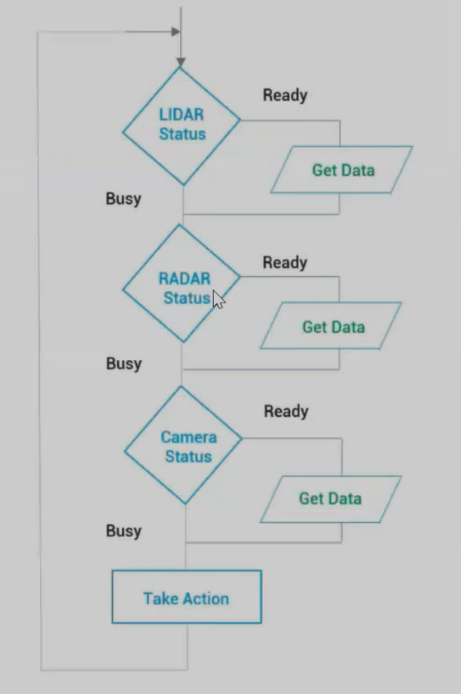
**Ưu điểm của RTOS**

* Độ ổn định và tin cây cao, nên có thể hoạt động trong thời gian dài mà không cần quá nhiều sự can thiệp của con người.
* Hiệu suất tốt hơn cùng mức tiêu thụ bộ nhớ thấp nên không gây tiêu tốn quá nhiều tài nguyên hoặc RAM.
* RTOS gần như không có lỗi và nếu có xảy ra lỗi cũng dễ dàng phát hiện hơn.
* Có kích thước nhỏ và chi phí thấp.



## Sofware Flow trong hệ thống nhúng (luồng chương trình)

### Busy Wait



Ví du: Hệ thống xe có 3 thiết bị.

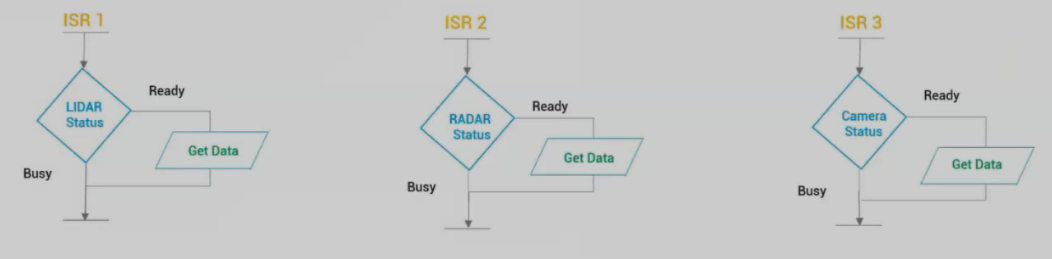
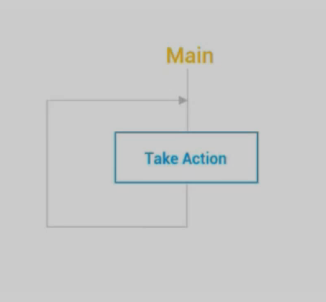
Theo sơ đồ, chương trình tổ chức các task kiểm tra các thiết bị theo thứ ự lần lượt. Cuối cùng sẽ thực hiện task Take Action dựa trên Data nhận được. Chỉ sử dụng được cho các hệ thống đơn giản.

## 4.2 Multi-Thread with ISR

Có thể tổ chức các Task nhận Data là các Task chạy nền. Data sẽ được gửi lên cho Task chính là Take Acion xử lý..

* Sử dụng Ngắt cho các Task nền, các Task này sẽ được chạy khi có Data hoặc chạy sau thời gian nhất định nhờ Timer.
* Task chính sẽ luôn chạy khi các Task khác không chạy.

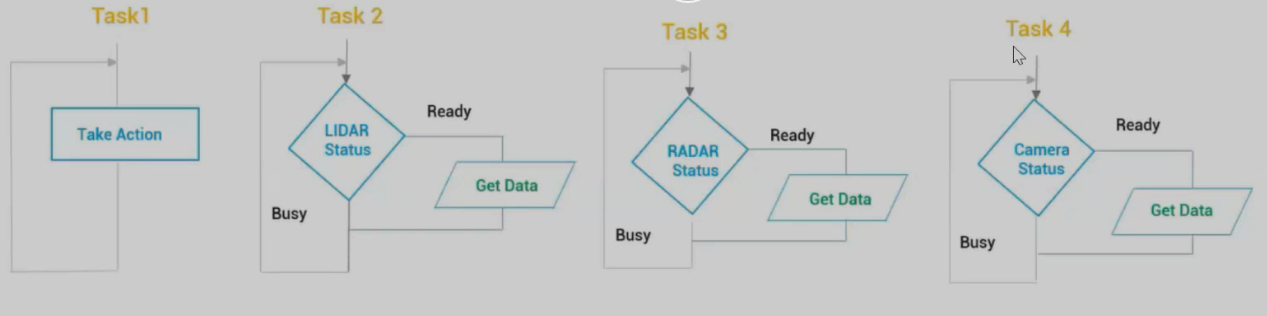
=> chỉ đáp ứng được ở mức Soft Real-Time.



## 4.3 RTOS

Để đáp ứng yêu cầu của RTOS, tất cả các Task đều sẽ được chạy như 1 Task chính và chạy gần như song song với nhau.

Mỗi Task lúc này sẽ như 1 chương trình riêng, ở đây là một function để thực hiện chức năng của chúng và thường sẽ có vòng lặp vô hạn riêng.



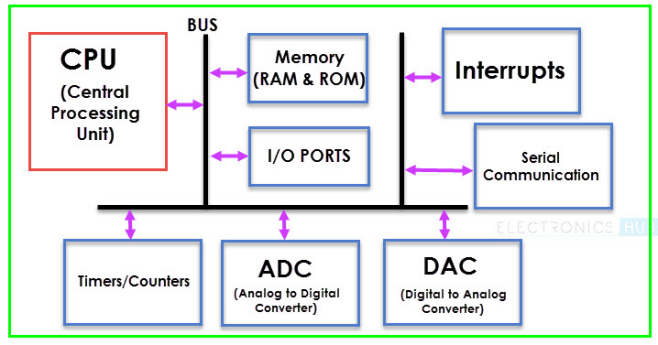
# Lesson 2: Cortex-M Overview Stack & Exception Handling

## Cortex-M Overview

Cấu trúc vi điều khiển STM32 cũng như các vi điều khiển cơ bản gồm:

* 1 bộ xử lý trung tâm - CPU.
* Bộ nhớ (RAM, ROM).
* Các ngoại vi (UART, SPI,…)
* Các I/O Ports.
* Một số bộ điều khiển ngắt.

Tất cả được kết nối qua các Bus.

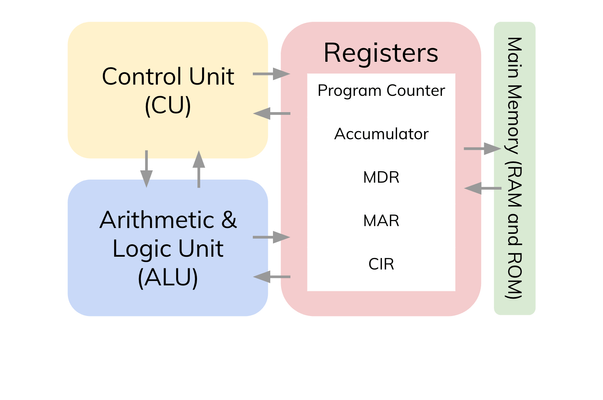


## CPU

CPU là trái tim của vi điều khiển, nhiệm vụ là tìm và nạp lệnh, giải mã và thực hiện lệnh.

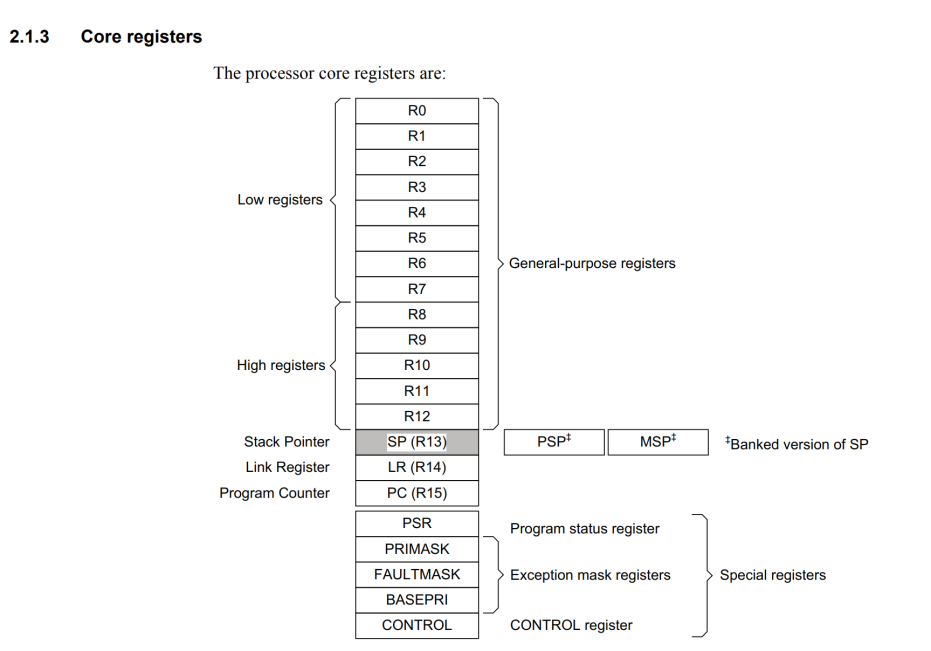
Cấu trúc chung của 1 CPU gồm:

* CU: điều khiển hoạt động của bộ xử lý.
* ALU: thực hiện tính toán số học vầ logic theo lệnh của CU.
* ALU: thực hiện tính toán trên bộ thanh ghi (Register Bank).



Đối với Cortex-M, bộ thanh ghi gồm 16 thanh ghi, R0-R15. Ngoài ra còn có các thanh đặc biệt.

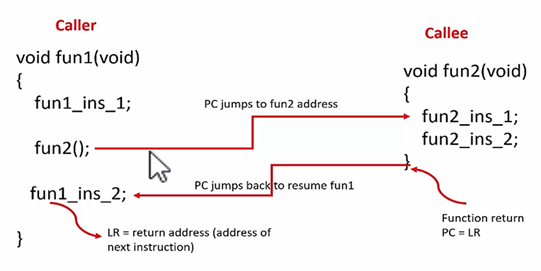
* R0-R12 là 13 thanh ghi đa dụng, dùng để tính toán.
* R14: Link Register, lưu vị trí trả về của 1 hàm.
* R15: PC, chưa địa chỉ lệnh tiếp theo sẽ được thực thi.
* R13: Stack Pointer, trỏ đến đỉnh Stack. Có 2 Stack Pointer. MSP và PSP.



Các thanh ghi từ R0 đến R12 là các thanh ghi mục đích chung (General Purpose Register), có tác dụng trong quá trình xử lý data, tính toán. Tức các phép tính toán, toán tử, toán hạng, kết quả,… sẽ được ALU tối ưu để đặt trên những thanh ghi này (hoặc người dùng lập trình - ngôn ngữ Assembly).

* **R13(SP - Stack Pointer)** trỏ đến một vùng nhớ đặc biệt gọi là Stack. Thanh ghi này có thể là MSP (Main SP) hoặc PSP (Process SP), dùng cho các Task trong RTOS.
* **R14(LR - Link Register)** là thanh ghi chứa địa chỉ trả về của các lệnh rẽ nhánh (ví dụ như khi gọi 1 hàm thì LR sẽ chứa địa chỉ của lệnh tiếp theo sau khi gọi hàm đó trả về).
* **R15(PC - Program Counter)** sử dụng để điều khiển quá trình thực thi các lệnh trong chương trình, PC sẽ chưa địa chỉ của lệnh tiếp theo mà Processor sẽ xử lý.

Một ví dụ để làm rõ hơn về hoạt động của LR VÀ PC:



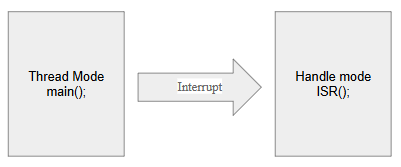
Ví dụ **function call:** Khi hàm **fun1()** gọi tới **fun2()** thì:

* PC sẽ nhảy đến địa chỉ của hàm **fun2()**.
* Lúc này LR sẽ chứa địa chỉ của lệnh tiếp theo, tucws là lệnh **fun1\_ins\_2()**;
* Và sau khi thực hiện xong hàm **fun2()** thì PC sẽ lấy giá trị của thanh ghi LR để trỏ đến lệnh tiếp theo **fun1\_ins\_2()** để tiếp tục thực hiện.

(Tham khảo tại <https://www.laptrinhdientu.com/2021/08/Core3.html>)

**Vị xử lý ARM Cortex-Mx hoạt động ở 2 chế độ:**

* Thread mode: tất cả các code chương trình viết trong main() sẽ được thực thi dưới chế độ Thread mode của vi xử lý. Mặc định, vi xử lý luôn bắt đầu ở chế độ Thread mode.
* Handler mode: tất cả các xử lý ngoại lệ (System Exception) và ngắt (Interrupt) sẽ được thực thi trong Handler mode của vi xử lý. Khi vi xử lý gặp System Exception hoặc Interrupt, chuyển từ Thread mode thành Handler mode và trình phục vụ tương ứng sẽ được thực thi ở Handler mode.

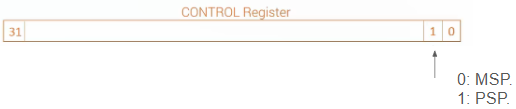


## MSP & PSP

Cortex-M có 2 SP:

* Main Stack Pointer (MSP): là SP mặc định, được sử dụng ở cả Thread mode và Handler mode. MSP được sử dụng trong chương trình chính và cả xử lý ngắt.
* Process Stack Pointer (PSP): là 1 SP thay thế, nó chỉ hoạt động ở Thread mode. PSP thường được sử dụng cho hoạt động của các Task trong hệ thống.

Trong Thread mode, bit[1] của thanh ghi CONTROL quy định SP nào sẽ được sử dụng:

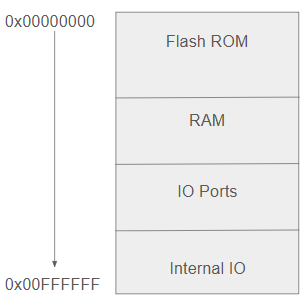


## Stack

Cortex-M Memory Map

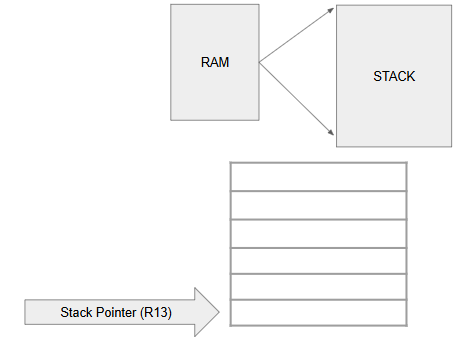
Các vi điều khiển Cortex-M đều được tổ chức bộ nhớ giống nhau, chỉ khác ở kích thước từng vùng.

Vùng nhớ bắt đầu ở địa chỉ 0x00000000 địa chỉ sẽ tăng theo chiều hướng xuống.



**Stack:**  được cấp phát trong RAM, là 1 vùng nhớ hoạt động theo kiểu LIFO. Vì kiến trúc là 32bit nên Stack sẽ hoạt động theo kiểu 32bit, mỗi ô nhớ cách nhau 4byte địa chỉ.

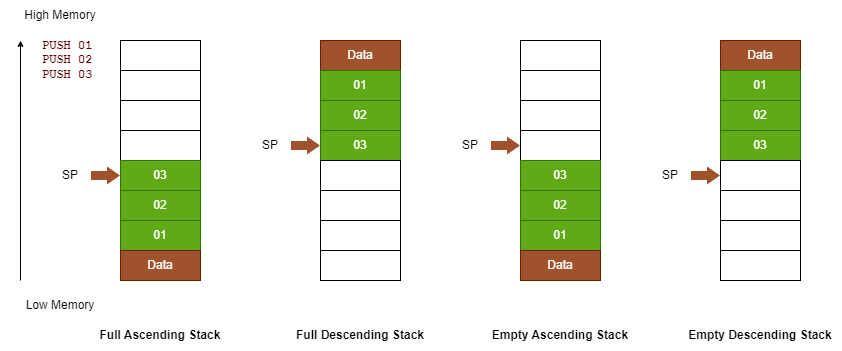
R13, Stack Pointer luôn trở tới địa chỉ đỉnh Stack.



**Stack Manipulation**

2 thao tác chính trên Stack là PUSH & POP.

* PUSH: đưa dữ liệu vào Stack.
* POP: lấy dữ liệu ra khỏi Stack.



Cortex-M sử dụng full descending stack. Tức là đáy của Stack sẽ là địa chỉ lớn nhất.

Khi PUSH data vào Stack, SP sẽ giảm 4 và data sẽ được thêm vào vị trí đó.

Ví dụ:

PUSH(R2); (R2 = 0x09)

SP = SP - 4;

\*(SP) = R2.

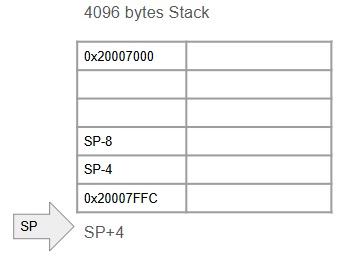
Khi POP dữ liệu, data tại SP sẽ được lấy ra khỏi Stack, sau đó SP sẽ tăng lên 4 để trỏ đến đỉnh Stack (Mới).

Ví dụ:

POP(R0); (\*(SP) = 0x08)

\*(R0) = \*(SP) = 0x08;

SP = SP + 4;

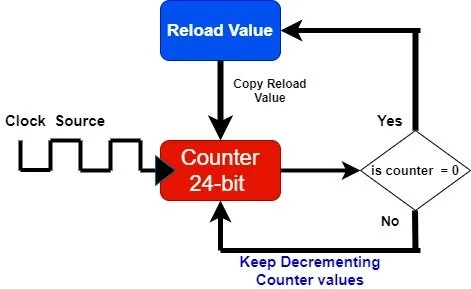


## SysTick

SysTick là 1 timer của hệ thống, không giống như các ngoại vi Timer, SysTick Timer là 1 Timer độc lập nắm trong lõi Cortex.

SysTick là Timer 24-bit, đếm ngược xảy ra ngắt khi giá trị đếm bằng 0 và tự nạp lại giá trị đếm ban đầu.

Trong RTOS, SysTick thường dùng để tạo tín hiệu đồng bộ các Task.



**Một số thanh ghi sau để cấu hình và điều khiển Systick Timer.**

**SYST\_CSR - Systick Control & Status Register:** cấu hình điều khiển và trạng thái.

* Bit[16] - COUNTFLAG: cờ này được tự động set lên 1 khi bộ đếm trán (đếm về 0).
* Bit[2] - CLKSOURCE: chọn nguồn cấp xung clock cho timer.
  + 0 = External Clock = AHB/8.
  + 1 = Processor Clock = AHB.
* Bit[1] - TICKINT: cho phép ngắt Systick nếu bit này bằng 1.
* Bit[0] - ENABLE: cho phép bộ đếm hoạt động.
  + 0 = DISABLE.
  + 1 = ENABLE.

Khi được ENABLE, bộ đếm sẽ lấy giá trị đã cài đặt sẵn trong thanh ghi Reload và bắt đầu đếm ngược.

**SYST\_RVR - Systick Reload Value Register:**

* Thanh ghi này chưa giá trị nạp lại để bắt đầu đếm.
* Vì SysTick là timer 24bit, nên chỉ sử dụng 24bit thấp.
* Giá trị từ 0x00000001 - 0x00FFFFFF.

**SYST\_CVR -Systick Current Value Register:**

* Đây là thanh ghi chứa giá trị đếm của SysTick.
* Thanh ghi này sẽ bắt đầu với giá trị lấy từ thanh ghi SYST\_RVR và đếm ngược về 0.

**Cấu hình SysTick.**

* Đặt giá trị đếm ban đầu - Reload Value bằng thanh ghi SYST\_RVR.
* Xóa giá trị đếm hiện tại trên thanh ghi SYST\_CVR.
* Cấu hình hoạt động cho Systick Timer bằng thanh ghi Control SYST\_CSR.
  + Chọn nguồn cấp xung Clock cho SysTick bằng Bit[2] - CLKSOURCE.
  + Cho phép ngắt Systick (nếu sử dụng ngắt) bằng Bit[1] - TICKINT.
  + Cho phép bộ đếm Systick hoạt động bằng Bit[0] - ENABLE.

Hàm phục vụ ngắt SysTick\_Handler() sẽ tự động được gọi khi SysTick đếm về 0.

Deylay(): ứng dụng cơ bản nhất của SysTick lầ tạo delay().

#define MAX\_DELAY 0xFFFFFFFF

\_\_IO uint32\_t g\_curr\_tick;

\_\_IO uint32\_t g\_curr\_tick\_p;

\_\_IO uint32\_t tick\_freq =1;

void Tick\_Increment(void){

g\_curr\_tick += tick\_freq;

}

void SysTick\_Handler(){

Tick\_Increment();

}

uint32\_t get\_tick(void){

\_\_disable\_irq();

g\_curr\_tick\_p = g\_curr\_tick;

\_\_enable\_irq();

return g\_curr\_tick\_p;

}

void delay(uint32\_t delay){

uint32\_t tick\_start = get\_tick();

uint32\_t wait = delay\*1000;

if(wait <MAX\_DELAY){

wait += (uint32\_t)(tick\_freq);

}

while((get\_tick()- tick\_start) < wait){}

}

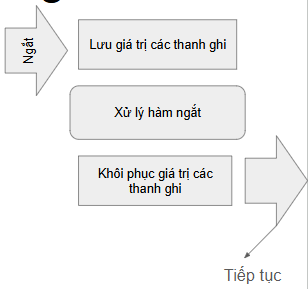
## Exception Handling

Với Cortex-M, khi 1 exception xảy ra (khi có sự kiện Ngắt):

* Gía trị trạng thái chương trình được lưu lại.
* Bộ xử lý chuyển đến chương trình phục vụ xử lý exception.
* Sau khi xử lý exception, các giá trị trạng thái sẽ được khôi phục.
* Chương trình tiếp tục tại nơi mà nó đã tạm dừng để xử lý exception.

-> Việc lưu lại giá trị trạng thái chương trình là việc lưu các giá trị thanh ghi trong CPU.

-> Khôi phục giá trị tương ứng với việc khôi phục lại giá trị cho từng thanh ghi tương ứng.

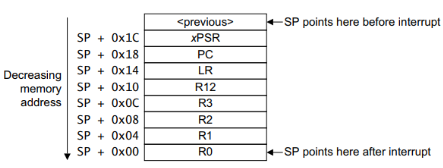


Video: <https://www.youtube.com/watch?v=mNUH0O_4Zn8>

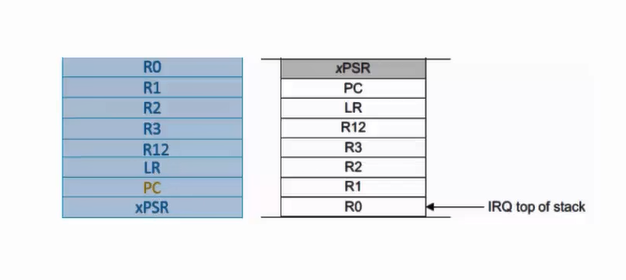
## Stack Frame

Các thanh ghi sẽ được bộ xử lý PUSH vào Stack khi exception xảy ra. Bao gồm 8 thanh ghi sắp xếp theo cấu trúc, gọi là Stack Frame, chứa các thông tin:

* xPSR: chứa thông tin hiện tại của chương trình.
* R0, R1, R2, R3, R12.
* LR: chứa địa chỉ lệnh tiếp theo khi làm hiện tại trả về.
* PC: chứa địa chỉ lệnh tiếp theo sau khi hoàn tất exception. Có nghĩa là lệnh đáng lẽ sẽ được thực hiện nếu không có exception.



Khi Stack Frame lưu vào bộ nhớ và khôi phục hoàn toàn tự động, việc sắp xếp trong Stack như hình sau:



Điều này dẫn đến một khả năng: Nếu ta thay đổi thông tin của Stack Frame trong khi xử lý exception, bộ xử lý sẽ cập nhật dữ liệu mới này vào các thanh ghi sau khi thoát exception.

Video: <https://www.youtube.com/watch?v=oQ8Mc2TuYPU&t=4s>

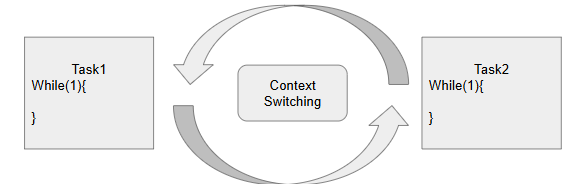
## Manual Context Switching

Quá trình lưu trữ, thay đổi và khôi phục lại Stack Frame mới gọi là Context Switch.

Ứng dụng quá trình này, thay đổi giá trị PC, ta có thể thay đổi việc chạy giữa các task với nhau trong RTOS.

Từ đó có thể phát triển ứng dụng giúp chạy các task đồng bộ với nhau.

* Sử dụng SysTick để tạo ngắt định kì -> bên trong hàm ngắt xử lý để chon task thực thi -> sau đó cập nhật.



# Lesson 3: Thread, Scheduler and Kernel

## Thread (Task)

### Distinguish 3 defintions

* Multiprogramming: system executes many programs sequentially.
  + Một quá trình thực thi trong hệ thống máy tính chủ yếu yêu cầu hai thứ, CPU time và I/O time. CPU time là thời gian CPU thực hiện một tiến trình (process) và I/O time là thời gian mà tiến trinh thực hiện cho các thao tác I/O chẳng hạn như một số thao tác trên file (tệp) như read và write. Nói chung, hệ thống máy tính của mình mong muốn thực thi một số quy trình cùng một lúc. Nhưng điều nay là không thể. Chúng ta chỉ có thể chạy một process tại một thời điểm trong một bộ xử lý (a processor). Nhưng điều nay có thể dẫn tới một số vấn đề.
  + Giả sử chúng ta có một hệ thống xử lý đơn và có 5 tiến trình P1, P2, P3, P4 và P5 sẽ được CPU thực thi. Vì CPU chỉ có thể thực thi một tiến trình tại một thời điểm, nên nó bắt đầu với tiến trình P1 và sau một thời thực thi tiến trình P1, tiến trình P1 yêu cầu một số theo tác I/O (I/O operation). Vì vậy, nó rời khỏi CPU và bắt đầu thực hiện thao tác I/O đó. Bây giờ, CPU sẽ đợi tiến trình P1 (process P1) quay trở lại để thực thi và CPU sẽ ở trạng thái không hoạt động trong khoảng thời đó. Nhưng đồng thời, các tiến trình khác như P2, P3, P4 và P5 đang chờ thực thi ở đó. CPU của chúng ta đang ở trạng thái không hoạt động và đây là một điều rất tốn kém. Vậy tại sao phải giữu CPU ở trạng thái không hoạt động? Những gì chúng ta có thể làm là, nếu tiến trình P1 muốn thực hiện một số thao tác I/O thì hãy để tiến trình P1 thực hiện công việc I/O và đồng thời, CPU sẽ được cấp cho tiến trình P2 và nếu tiến trình P2 cũng có yêu cầu một số thao tác I/O thì CPU sẽ được giao để xử lý tiến trình P3,… Điều này được gọi là **Context Switching.** Khi công việc I/O được tiến trình thực hiện thì CPU có thể tiếp tục hoạt động của tiến trình đó (tức là process P1 nad P2) và bằng cách làm như vậy, CPU sẽ không bao giwof chueyenr sang trạng thái không hoạt động. Khái niệm sử dụng CPU hiểu quả này là được gọi là **Multiprogramming.**

=> Vì vậy, trong Multi programming, CPU thực thi một phần nào đó của một chương trình, sau đó là một phần chương trình khác. Bằng cách này, CPU sẽ không bao giờ chuyển sang trạng thái không hoạt động (idle) trừ khi không có tiến trình nào sẵn sàng thực thi tại thời điểm Context Switching.

Tất cả các hệ điều hành hiện đại như MS Windows, Linux, v.v. đều là hệ điều hành đa chương trình.

VD: Consider 4 programmes say P1,P2,P3,P4 are in the main memory and need the CPU to get processed. Let us get this through a conversation

CPU : Come on P1,I will execute you .

( P1 executes for sometime and need to access I/O )

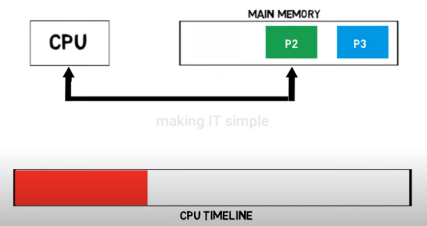
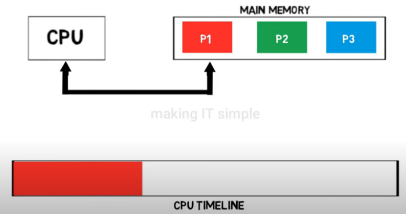
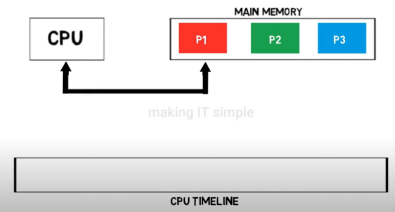
P1 : Hey CPU, I need I/O.

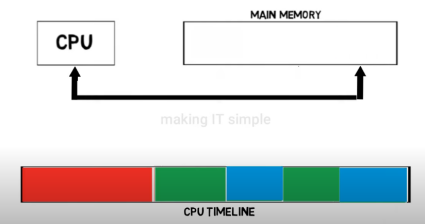
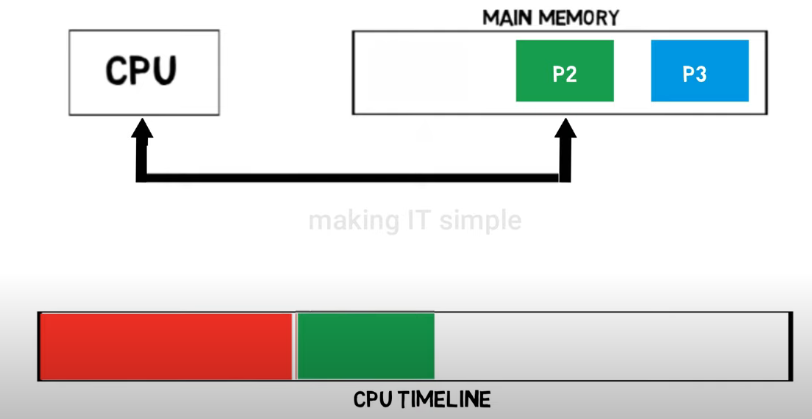
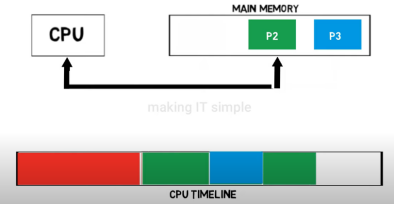
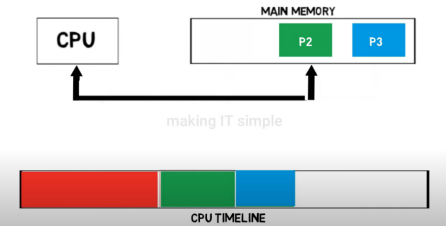
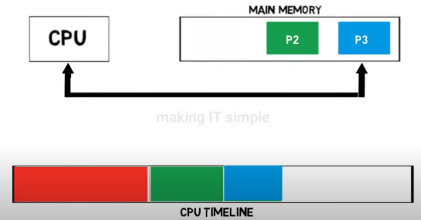
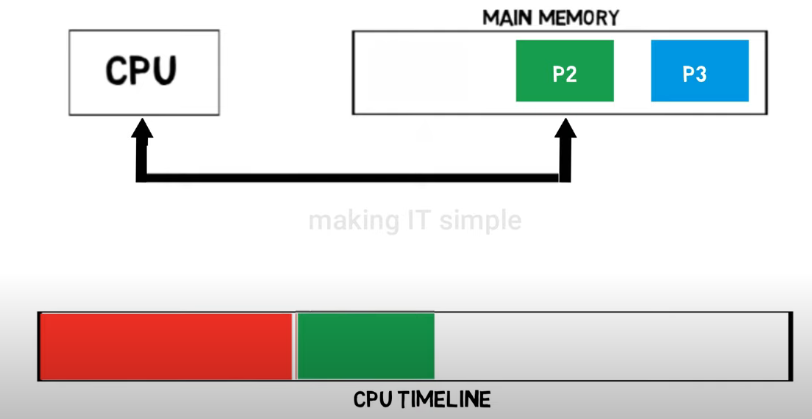
CPU : Okay go for it. Come on P2 , Now I will execute you.

( P1 returns to CPU after its I/O completion ).

CPU : P2 You need to wait as P1 got returned ,Again I will execute You Once if P1 gets processed or goes for I/O.

=> Ở trên, khi P1 thực hiện I/O, CPU không hoạt động. Nó sẽ thực thi tiến trình tiếp theo trong bộ nhớ chính, tức là P2 cho đến khi P1 quay trở lại. Khi công việc P1 hoàn thành, P2 sẽ được thực thi và quá trình này sẽ tiếp tục..





**Ưu điểm của đa chương trình:**

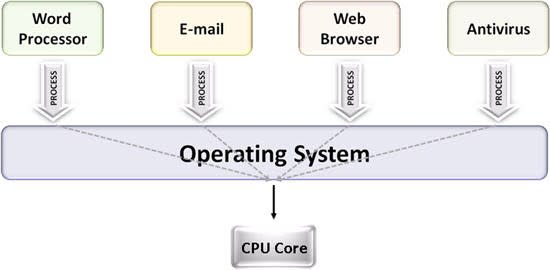
* Cần CPU đơn để thực hiện.
* Chuyển đổi bối cảnh giữa quá trình. Context switching
* Chuyển đổi xảy ra khi quá trình hiện tại trải qua trạng thái chờ.
* Thời gian nhàn rỗi của CPU giảm.
* Sử dụng tài nguyên cao.
* Hiệu suất cao.

**Nhược điểm của đa chương trình:**

* Vì bạn phải thực hiện Context Switching, nên bạn cần có một số kỹ thuật lập lịch quy trình để cho CPU biết quy trình nào cần thực hiện và quy trình đó khó thực hiện.
* Ở đây, CPU đang thực thi một số phần của một quy trình, sau đó là một phần của quy trình khác, v.v. Vì vậy, trong trường hợp này, bộ nhớ sẽ được chia thành các phần nhỏ vì mỗi quá trình yêu cầu một số bộ nhớ và điều này sẽ dẫn đến sự phân mảnh bộ nhớ. Vì vậy, sẽ không có hoặc có ít bộ nhớ liên tục khả dụng.
* Multitasking: system executes many programs concurrently.
* Nếu CPU được phân bổ cho một tiến trình tốn nhiều thời gian như vậy thì các tiến trình khác sẽ phải chờ quá trình đó được thực thi và điều này sẽ dẫn đến việc các tiến trình phải đợi lâu để được phân bổ tài nguyên.
  + VD: nếu tiến trình P1 chiếm 20s CPU time và CPU được phân bổ cho P1. Bây giờ nếu có quá trình P2 nào đó yêu cầu 1s CPU time thì P2 phải đợi 20s bất chấp thực tế là nó chỉ yêu cầu 1s CPU time.
* Những gì chúng ta có thể làm ở đây là, chúng ta có thể đặt lượng tử thời gian và CPU sẽ chỉ được cấp cho mỗi tiến trình trong khoảng thời gian nào đó và sau đó CPU sẽ được cấp cho một số tiến trình khác đã sẵn sàng để thực thi. Vì vậy, trong ví dụ trên chúng ta nên chỉ định mức thời gian lương tử là 2s, thì ban đầu tiến trình P1 sẽ được cấp CPU trong 2s và sau đó nó sẽ được cấp cho tiến trình P2 . Tiến trình P2 sẽ hoàn thành việc thực thi sau 1s và sau đó CPU sẽ được cấp quyền xử lý lại P1. Vì không có tiến trình nào khác có sẵn để thực thi nên tiến trình P1 có thể tiếp tục thực thi trong thời gian còn lại, tức là 18s. Điều này được gọi là **time - sharing.** Và khái niệm time-sharing giữa các tiến trình khác nhau được gọi là **Multitask**.
* **Multitask** là **Multiprogramming** với **time-sharing**. Ở đây, vieejcc chuyển đổi giữa các tiến trình nhanh đến mức tạo ra ảo giác rằng tất cả các tiến trình đang được thực thi cùng một lúc. Để thực hiện Multitask, trước hết phải cần có Multiprogramming ba thứ 2 cần có sự time-sharing.

VD: Giả sử bạn muốn nghe nhạc mà muốn viết code cùng lúc. Điều này đạt được bằng cách thực hiện đa nhiệm (multitask)

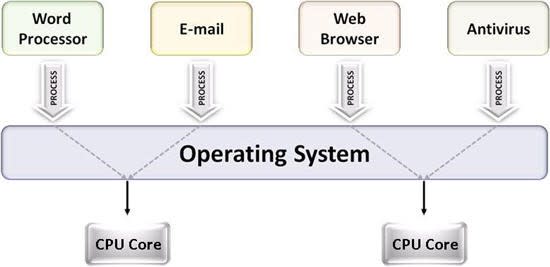
=> **Đa nhiệm trong hệ thống lõi đơn (single core OS):** chúng ta nghĩ rằng hệ thống của bạn thực sự thực thi nhiều chương trình đồng thời không? Rõ ràng là không! Những gì hệ điều hành của bạn thực sự làm là nó thực thi từng chương trình trong một khoảng thời gian ngắn và chuyển sang chương trình khác.



* Multiprocessing: system has two or more cores, so that it can execute many programs in parallel.
  + Chúng ta đều biết rằng trong một hệ thống đơn xử lý, bộ xử lý chỉ có thể thực hiện một tiến trình tại một thời điểm. Tuy nhiên, khi hệ thống của bạn có rất nhiều công việc cần thực hiện và một bộ xử lý là quá ít để hoàn thành tất cả các công việc đó trong khoảng thời gian yêu cầu, thì chúng ta có thể sử dụng nhiều bộ xử lý trong cùng một hệ thống.

=> **Trong một hệ thống đa xử lý, hai hoặc nhiều bộ xử lý có mặt trong cùng một máy tính, chia sẻ bus hệ thống, bộ nhớ và các thiết bị I/O khác.**

* Giả sử, chúng ta có 5 tiến trình P1, P2, P3, P4 và P5. Trong một hệ thống đơn xử lý, chỉ có một tiến trình có thể được thực thi tại một thời điểm và sau khi tiến trình đó được thực thi xong, tiến trình tiếp theo sẽ được thực thi và cứ tiếp tục như vậy. Nhưng trong một hệ thống đa xử lý, các tiến trình khác nhau có thể được phân công cho các bộ xử lý khác nhau, và điều này làm giảm thời gian thực thi tổng thể của các tiến trình trong hệ thống. Một hệ thống hai bộ xử lý có thể thực thi hai tiến trình cùng một lúc, trong khi một hệ thống bốn bộ xử lý có thể thực thi bốn tiến trình cùng một lúc.

=>Multiprocessing and Multitask

**Ưu điểm của đa xử lý**

* Vì có nhiều hơn một bộ xử lý hoạt động cùng một lúc, nên có được nhiều công việc thực hiện trong thời gian ngắn hơn. Lưu lượng xử lý sẽ được tăng cường. Bạn có thể đọc thêm về lưu lượng xử lý từ đây.
* Chúng ta có nhiều hơn một bộ xử lý, vì vậy nếu một bộ xử lý không hoạt động thì công việc có thể được thực hiện với sự trợ giúp của các bộ xử lý khác. Điều này, lần lượt, tăng tính tin cậy.
* Nếu bạn đưa ra nhiều công việc trên một bộ xử lý thì sẽ dẫn đến tiêu hao pin nhiều hơn. Nhưng nếu công việc được chia thành các bộ xử lý khác nhau thì sẽ cung cấp hiệu suất pin tốt hơn.
* Đa xử lý là một ví dụ về xử lý song song thực sự, tức là nhiều tiến trình được thực hiện cùng một lúc.

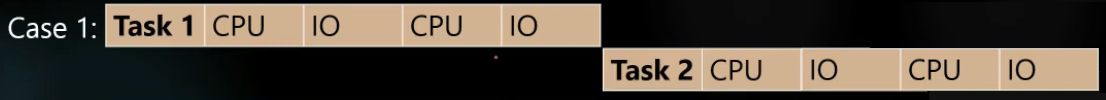
**Nhược điểm của đa xử lý**

* + Khi có nhiều bộ xử lý hoạt động vào một thời điểm cụ thể, sự phối hợp giữa chúng trở nên rất phức tạp.
  + Bởi vì các bus, bộ nhớ và thiết bị I/O được chia sẻ. Vì vậy, nếu một số bộ xử lý đang sử dụng một số thiết bị I/O thì bộ xử lý khác phải chờ đợi lượt của mình, và điều này sẽ dẫn đến giảm lưu lượng xử lý.
  + Để có hoạt động hiệu quả của tất cả các bộ xử lý cùng một lúc, chúng ta cần có một bộ nhớ chính lớn, và điều này, lần lượt, làm tăng chi phí.

#### Concurrent/ parallel processing demand

There are 2 motivation for modern OS generally supports multitasking and multiprocessing:

* Improving CPU usage.
  + The CPU speed (GHz) is much faster than I/O speed (MHz).
  + While waiting for the I/O respond, the CPU executes the other task.





* Increasing processing speed.
  + If the CPU has 2 cores or more, it can execute 2 processes or more in parallel, so that we can save the processing time.
  + Example: calculate this kq = a\*b + c\*d

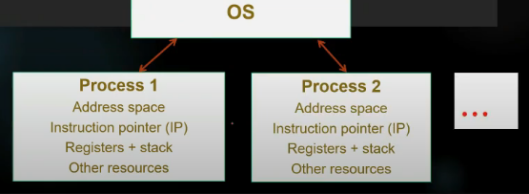
Method 1: Method 2:

|  |  |
| --- | --- |
| 2 CPU | |
| X = a\*b | Y = c\*d |
| Kq = X + Y |  |

|  |
| --- |
| 1 CPU |
| X = a\*b |
| Y = c\*d |
| Kq = X + Y |

#### Process and Multi process model

* The program stores in auxiliary memory, only contains instructions.
* Process is the program that is loaded into the **main memory**, and given the CPU and the resources to execute. VD: Task manager (window os)
* Processers communicate with each other through the OS. (các tiến trình không truền nhận trực tiếp với nhau phải thông qua OS)
* Process has its own resources, doesn’t interfere with the other. (tài nguyên của tiến trình là riêng biệt không liên quan với nhau độc lập).



* The parent process send a “system call” for the OS, then OS create the child process. (Only OS can create process).



#### Thread and multi-thread model

What is thread ?

* Thread is a sub-process or tiny process which runs inside a process.
* Threads in the same process share the common address space, which means threads can share the process’s global resource.
* Thread A can access Thread B’s stack (theory).

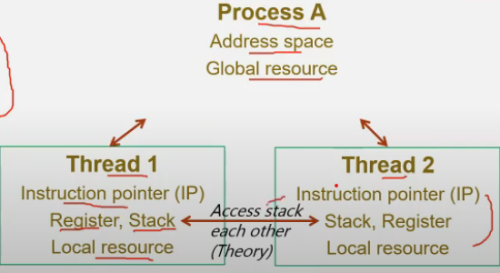
What is multi-thread?

* Multi-threading is the term used to run multiple thread inside a single process in order to execute multiple tasks at the same time.

*=> Thread share CPU with each other as well as the processes do.*

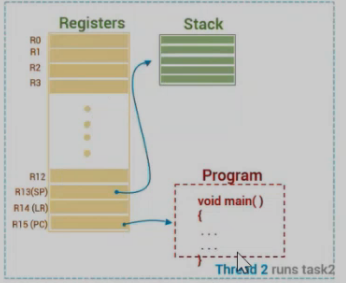
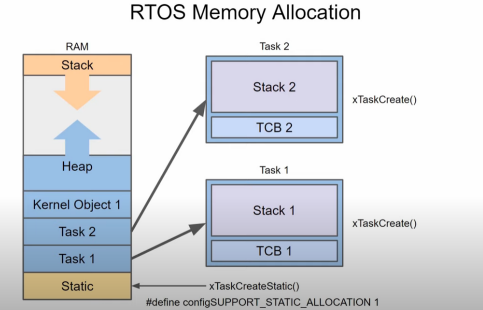
*=> Threads can be further subdivided into multiple child processes.*

*=> When new thread is created, the process requests OS to allocate more resources (memory,…) to create thread. (as the way, the program allocates the memory for pointer)*



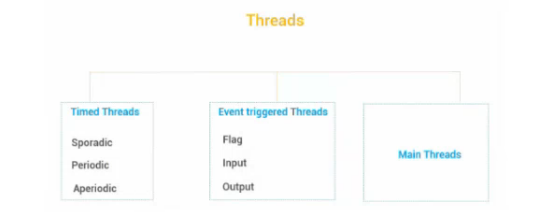
**Thread gồm:**

* Register back.
* Stack.
* Chương trình để thực thi.



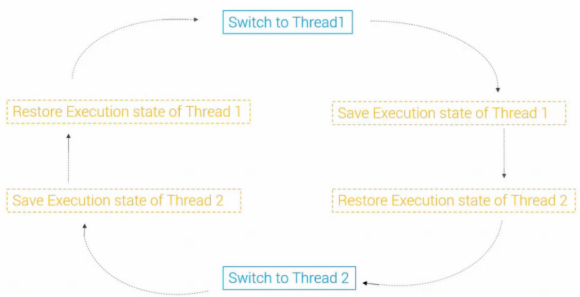
=> STM32 chỉ có 1 core với 1 register bank duy nhất => phải liên tục chuyển đổi việc sử dụng thanh ghi giữa các Task (thread)

**Phân loại Thread:**

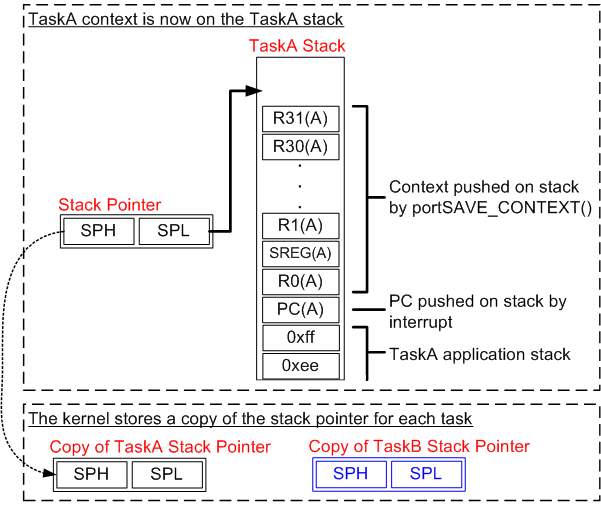
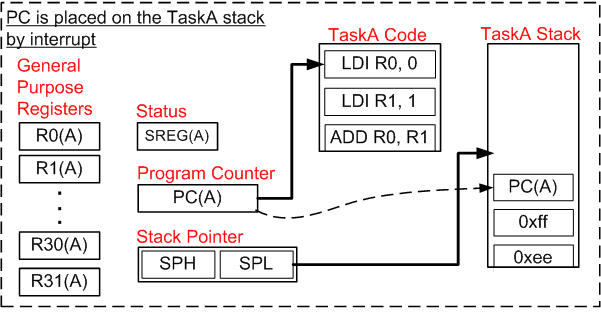
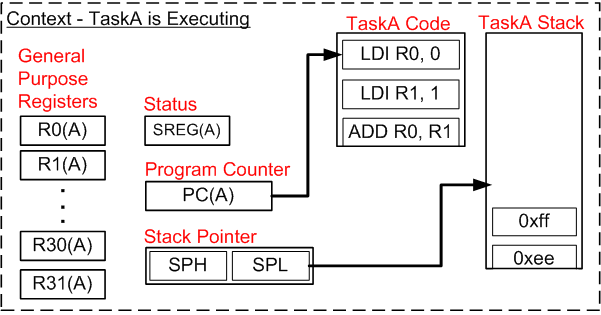


* Timed threads:
  + Các tác vụ chạy khi có sự cố.
  + Các tác vụ chạy định kì.
  + Các tác vụ không đoán trước được.
* Event triggered threads:
  + Các tác vụ được khởi chạy khi có 1 cờ nào đó được set hay reset.
  + Các thread chạy khi có input/output.
* Main threads:
  + Các tác vụ chạy khi hệ thống rảnh.

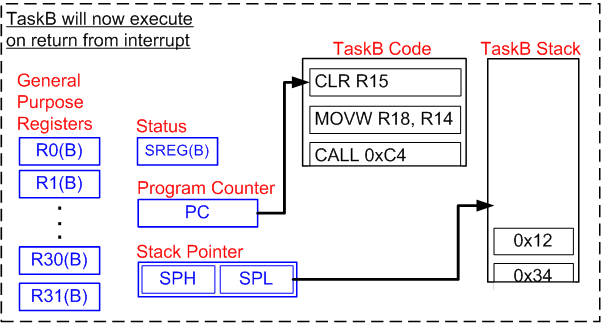
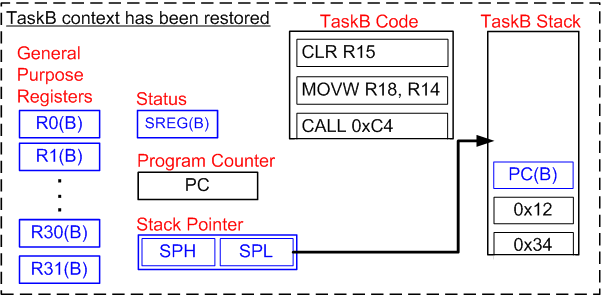
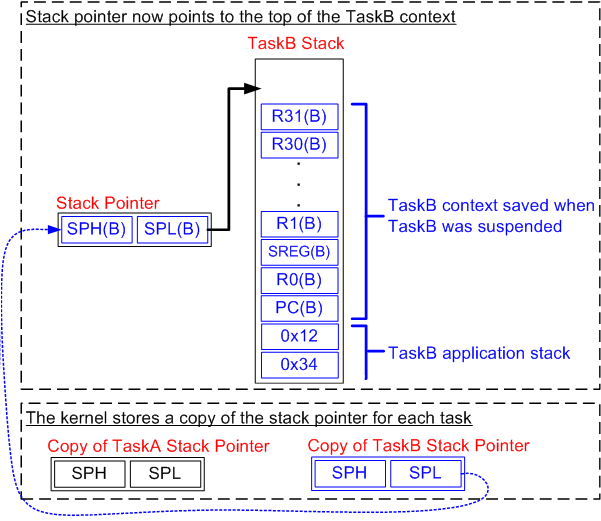
**Thread Switching Process**



* ***Save execution state of a thread.***



* ***Restore execution state of a thread.***



#### Thread control block (TCB)

* Để lưu thông tin 1 thread thì cần có 1 cấu trúc dữ liệu.
* TCB là một struct chứa thông tin của mỗi thread. Gồm 2 thông tin chính:
  + Pointer to stack.
  + Pointer to next Thread.
* Ngoài ra còn có các biến thông tin Thread
  + Status.
  + ID.
  + Priority.
  + …

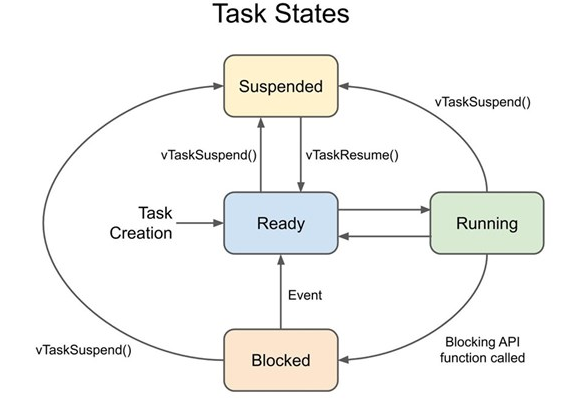
#### Thread Scheduler

Thread scheduler đảm nhiệm việc dừng 1 thread đang chạy và chọn 1 thread khác cho CPU thực hiện dựa trên thuật toán.

=> là 1 hàm có chức năng chạy các thread theo thứ tư lần lượt từng thread một.

***Trong Scheduler mỗi tác vụ sẽ có 3 trạng thái mặc định là:***

* Ready to run: Trạng thái chuẩn bị của tác vụ khi đã có đủ các tài nguyên để khởi chạy.
* Running: Trạng thái tác vụ đang thực thi.
* Blocked: Đây là trạng thái mà các tác vụ không đủ tài nguyên để xử lý sẽ được về trạng thái khóa.



***Trạng thái Ready***: để chuyển sang trạng thái Running, tác vụ phải ở trạng thái Ready. Khi mới được khởi tạo, tác vụ sẽ được đặt ở trạng thái Ready. Nếu tác vụ bị dừng lại bởi hệ điều hành để nhường quyền thực thi cho tác vụ khác khi nó hết thời gian cho phép giữa hai lần ngắt Systick, nó sẽ được đặt vào trạng thái Ready, trong trường hợp này nó sẽ sẵn sàng chuyển sang trạng thái Running.

***Trạng thái Blocked***: Ngoài trạng thái Running, tác vụ còn có thể đi vào trạng thái Blocked, khi đó tác vụ chờ một sự kiện xảy ra để đi vào trạng thái Ready. Có hai sự kiện như sau:

* Sự kiện thời gian: sự kiện này có thể là một khoảng thời gian trễ đã hết, hoặc tác vụ đạt đến một thời điểm tuyệt đối nào đó. Ví dụ như tác vụ đi vào trạng thái Blocked chờ một khoảng thời gian 10ms.
* Sự kiện đồng bộ: một tác vụ chờ một sự kiện được tạo ra từ ngắt hay từ tác vụ khác. Ví dụ một tác vụ đi vào trạng thái Blocked để chờ dữ liệu được gửi đến hàng chờ (queue) từ một ngắt.

Ta có thể chỉ định cho một tác vụ đi vào trạng thái Blocked để chờ sự kiện đồng bộ trong một khoảng thời gian time-out. Ví dụ 1 tác vụ bị Blocked để chờ dữ liệu gửi đến hàng đợi trong 10ms. Tác vụ này sẽ đi vào trạng thái Ready ngay khi có dữ liệu gử đến trong thời gian 10ms, hoặc sau thời gian 10ms mà không có dữ liệu nào gửi đến.

***Trạng thái Suspended:*** ở trạng thái Suspended, tác vụ không thể tự đi vào trạng thái Ready. Để đi vào trạng thái Suspended, hàm vTaskSuspend() được gọi từ tác vụ khác hay chính tác vụ đó. Để đưa một tác vụ ra khỏi trạng thái Suspended, hàm vTaskResume() được gọi từ tác vụ khác hoặc hàm vTaskResumeFromISR() được gọi trong một ngắt.

Trạng thái Suspended này ít khi được sử dụng khi lập trình.

#### Preemption (quyền ưu tiên)

Khi một thread chuyển từu Running sang Blocked, nó sẽ tự hoãn chính nó vì phải đợi 1 event.

Khi 1 thread chuyển từ Running sang Ready, tức là nó bị Scheduler bắt buộc nhường quyền thực thi cho thread khác, bị preempt bởi scheduler. Nhằm mục đích:

* Đảm bảo tính công bằng các thread.
* Giúp thread hoàn thành deadlines. =>

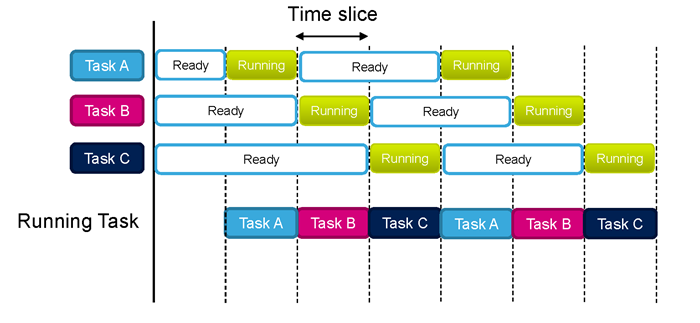
=> cần ngắt để tạo preemption.

#### Mỗi Scheduler sẽ có thuật toán riêng để quyết định trạng thái của các Thread. Có thể được phân loại gồm:

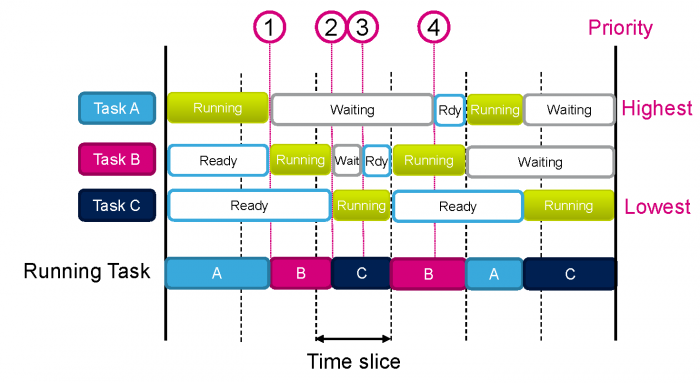
* Static & Dynamic:
  + Static: độ ưu tiên của thread được xác định trước khi hệ thống chạy.
  + Dynamic: độ ưu tiên của thread được xác định trong quá trình chạy.
* Preemptive & Non-Preemptive:
  + Preemptive: thread có thể bị hoãn do 1 thread khác có độ ưu tiên cao hơn hoặc do ngắt.
  + Non-preemptive: khi 1 thread chạy, nó có quyền hoàn tất thực thi trước khi CPU thực thi thread khác, trừ khi nó chủ động nhương CPU.

=> Scheduler có thể kết hợp giữa hai loại.

* Cooperative: giống với lập trình thông thường, mỗi task chỉ có thể thực thi khi task đang chạy dừng lại, nhược điểm của nó là task này có thể dùng hết tất cả tài nguyên của CPU.
* Round-robin: mỗi task được thực hiện trong thời gian định trước (time slice) và không có ưu tiên.



* Priority base: Task được phân quyền cao nhất sẽ được thực hiện trước, nếu các task có cùng quyền như nhau thì sẽ giống với round-robin, các task có mức ưu tiên thấp hơn sẽ được thực hiện cho đến cuối time slic.



* Priority-based pre-emptive: Các task có mức ưu tiên cao nhất luôn nhường các task có mức ưu tiên thấp hơn thực thi trước.



#### Schedule Criteria

Một Scheduler được đánh giá dựa trên các tiêu chí:

* Throughput: số lương task thực hiện được trong một đơn vị thời gian.
* Turnaround time: thời gian để hoàn thàn mỗi task. Từ lúc task xuất hiện đến lúc hoàn thành.
* Response time: thời gian từ khi yêu cầu đến khí nhận được phản hồi đầu tiên.
* Wait time: thời gian mỗi task phải chờ trong trạng thái Ready.
* CPU Utilization: % chu kì CPU khả dụng đang được dùng.

Một Scheduler tốt sẽ cần:

* Tối đa hóa throughput.
* Giảm turnaround time
* Giảm response tim.
* Tối ưu hóa CPU Utilization.

#### Các thuật toán Scheduler

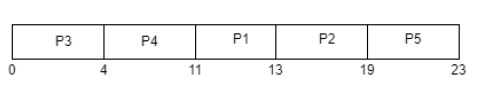
1. ***FCFS (First come - First serve)***

* Non-preemptive.
* Task nào vào Ready List first thì dùng CPU trước.

VD:

|  |  |  |
| --- | --- | --- |
| **Process ID** | **Arrival time** | **Burst time** |
| P1 | 2 | 2 |
| P2 | 5 | 6 |
| P3 | 0 | 4 |
| P4 | 0 | 7 |
| P5 | 7 | 4 |

Gantt chart:



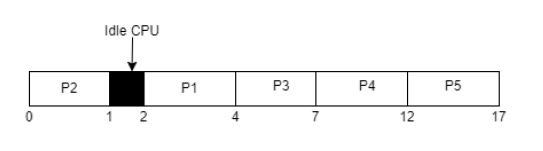
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Process ID** | **Arrival time** | **Burst time** | **CT** | **TAT=CT-AT** | **WT=TAT-BT** | **RT** |
| P1 | 2 | 2 | 13 | 13-2= 11 | 11-2= 9 | 9 |
| P2 | 5 | 6 | 19 | 19-5= 14 | 14-6= 8 | 8 |
| P3 | 0 | 4 | 4 | 4-0= 4 | 4-4= 0 | 0 |
| P4 | 0 | 7 | 11 | 11-0= 11 | 11-7= 4 | 4 |
| P5 | 7 | 4 | 23 | 23-7= 16 | 16-4= 12 | 12 |

* + Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)
  + Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds)

VD:

|  |  |  |
| --- | --- | --- |
| **Process ID** | **Arrival time** | **Burst time** |
| P1 | 2 | 2 |
| P2 | 0 | 1 |
| P3 | 2 | 3 |
| P4 | 3 | 5 |
| P5 | 4 | 5 |

Gantt chart:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Process ID** | **Arrival time** | **Burst time** | **CT** | **TAT=CT-AT** | **WT=TAT-BT** | **RT** |
| P1 | 2 | 2 | 4 | 4-2= 2 | 2-2= 0 | 0 |
| P2 | 0 | 1 | 1 | 1-0= 1 | 1-1= 0 | 0 |
| P3 | 2 | 3 | 7 | 7-2= 5 | 5-3= 2 | 2 |
| P4 | 3 | 5 | 12 | 12-3= 9 | 9-5= 4 | 4 |
| P5 | 4 | 5 | 17 | 17-4= 13 | 13-5= 8 | 8 |

* + Average Waiting time = (0+0+2+4+8)/5 = 14/5 = 2.8 time unit (time unit can be considered as milliseconds)
  + Average Turn-around time = (2+1+5+9+13)/5 = 30/5 = 6 time unit (time unit can be considered as milliseconds)

\*Trong khoảng thời gian CPU nhàn rỗi (không hoạt động), không có tiến trình nào được lên lịch để kết thúc nên trong thời gian này nó vẫn trống trong một thời gian ngắn.

(https://www.tutorialspoint.com/fcfs-scheduling)

1. ***Round Robin.***

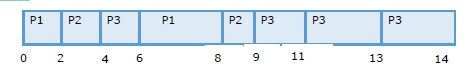
* Preemptive.
* Sử dụng việc chia sẻ thời gian. Mỗi thread sẽ có một thời gian thực hiện nhất định - timeslice (quanta).
* Khi một thread thực hiện hết một khoảng timeslice, Scheduler sẽ chuyển CPU cho thread khác.

**Example 1**−Given are the three processes P1, P2, and P3 with different arrival times, and burst time.

Quantum time =2.

|  |  |  |
| --- | --- | --- |
| **Process list** | **Arrival Time** | **Burst time** |
| P1 | 0 | 4 |
| P2 | 1 | 3 |
| P3 | 2 | 7 |

GANTT CHART: -1-3



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process list** | **Arrival time** | **Burst time** | **Completion time or closing time (CT)** | **Turn Around time(CT-AT)** | **Waiting time(TAT-BT)** |
| P1 | 0 | 4 | 8 | 8-0=8 | 8-4=4 |
| P2 | 2 | 3 | 9 | 9-2=7 | 7-3=4 |
| P3 | 4 | 7 | 14 | 14-4=10 | 10-7=3 |

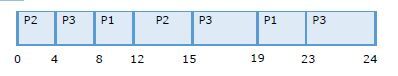
Thời gian hoàn thành là khoảng thời gian của mỗi quy trình khi nó hoàn thành việc thực thi thành công và được đưa ra khỏi hàng đợi, có thể được theo dõi bằng biểu đồ Gantt được cung cấp ở trên.

* Average WT = (4+4+3) / 3 = 3.66 units
* Average TAT = (8+7+10) / 3 = 8.33 units

**Example 2**−Given are the processes P1, P2, and P3 with different arrival times and burst times.

Quantum time = 4.

GANTT CHART



|  |  |  |
| --- | --- | --- |
| **Process list** | **Arrival Time** | **Burst time** |
| P1 | 2 | 8 |
| P2 | 0 | 7 |
| P3 | 1 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process list** | **Arrival time** | **Burst time** | **Completion time (CT)** | **TAT** | **WT** |
| P1 | 2 | 8 | 23 | 23-2=21 | 21-8=13 |
| P2 | 0 | 7 | 15 | 15-0=15 | 15-7=8 |
| P3 | 1 | 9 | 24 | 24-1=23 | 23-9=14 |

* + Average WT = (13+8+14) / 3 = 11.66 units
  + Average TAT = (21+15+23) / 3 = 19.66 units

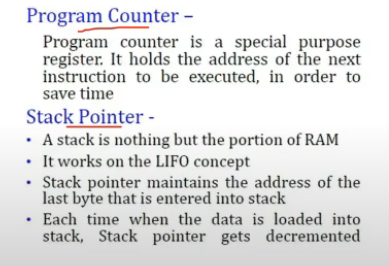
**Conclusion:** Thuật toán quay vòng được gọi là thuật toán ưu tiên vì quy trình ở trạng thái thực thi sẽ được bộ xử lý ưu tiên khi nó đạt đến thời gian lượng tử nhất định và được phục vụ sau dựa trên cơ sở ai đến trước được phục vụ trước. Phương pháp chuyển ngữ cảnh được sử dụng để lưu trữ trạng thái hiện tại của mọi tiến trình đã được ưu tiên.

(https://www.tutorialspoint.com/round-robin-scheduling-with-different-arrival-times)

#### Comparison: Process & Thread

The same: both of them contain:

* 1 instruction pointer (IP), (alias name: Program Counter(PC)).
* Stack memory, (stack pointer).
* Register.
* Other resources.



#### Comparison: Multi Process & Multi Thread

Difference:

|  |  |
| --- | --- |
| Multi-process model | Multi-thread model |
| Exchange information through OS. | Sharing the process’s global resources. (exchange though process) |
| Processes have a different address space. | Thread share the same common address space. |
| Slower in creating, switching, terminating. | Faster in creating, switching, terminating. |

\*Multi process model: can be multiprogramming, multitasking, multiprocessing.

#### Summary

* Process is the executing programing.
* To effectively use of CPU, multiprogramming, multitasking and multiprocessing should be incorporated into the system.
* The “multi” is organized by storing multi processes/threads in main memory in the time and coordinating the CPU back and forth between all of the processes/threads.
* Multi process model and multi thread model should be chosen depending on the specific situation.

Đa chương trình -

Nói một cách đơn giản, đó là khái niệm về một hệ thống máy tính chạy nhiều chương trình cùng một lúc.

Trong lập trình đơn, khi một công việc yêu cầu I/O và chuyển đến hàng đợi I/O, CPU sẽ không hoạt động vì nó không thể được phân bổ cho một quy trình khác trước khi quy trình hiện tại kết thúc quá trình thực thi của nó. Điều này dẫn đến lãng phí thời gian của CPU và làm giảm hiệu quả của hệ thống. Để khắc phục nhược điểm này, đa chương trình phát huy tác dụng.

Ý tưởng ở đây là, khi một tiến trình đang sử dụng hàng đợi I/O, CPU có thể được phân bổ cho một tiến trình khác đang chờ trong hàng đợi sẵn sàng để bắt đầu thực thi nó, do đó giảm thời gian nhàn rỗi của CPU. Mục tiêu cuối cùng ở đây là giữ cho CPU luôn bận rộn miễn là có các tiến trình được thực thi. Bằng cách này, nhiều chương trình có thể được thực thi trong một bộ xử lý.

Đa xử lý -

Một hệ thống máy tính sử dụng nhiều CPU cùng một lúc, tức là có nhiều bộ xử lý trong hệ thống. Bây giờ vì có nhiều bộ xử lý nên nhiều quy trình có thể được thực thi cùng một lúc.

Ưu điểm chính ở đây là hoàn thành được nhiều công việc hơn trong khoảng thời gian ngắn hơn. Loại hệ thống này được sử dụng khi cần tốc độ rất cao để xử lý khối lượng dữ liệu lớn.

Điều này cũng làm tăng độ tin cậy, nếu một bộ xử lý bị lỗi thì các bộ xử lý khác không dừng lại, chỉ có tốc độ giảm, tức là hệ thống chậm lại một chút.

Sự khác biệt giữa Đa chương trình và Đa xử lý -

Đa xử lý đang thực thi nhiều quy trình trên nhiều bộ xử lý, trong khi đó, đa chương trình đề cập đến khái niệm thực thi đồng thời nhiều chương trình bằng cách sử dụng một CPU.

Đa xử lý xảy ra bằng phương pháp xử lý song song, trong khi đó, đa chương trình yêu cầu chuyển đổi ngữ cảnh (chuyển đổi giữa hai quy trình).

Đa nhiệm —

Đề cập đến việc thực hiện nhiều nhiệm vụ cùng một lúc.

ví dụ- phát một số bản nhạc, chỉnh sửa tài liệu, lướt google cùng một lúc, đây là tính năng đa nhiệm.

Sự khác biệt chính với đa chương trình là đa chương trình hoạt động dựa trên khái niệm chuyển đổi ngữ cảnh, trong khi đó, đa nhiệm dựa trên việc chia sẻ thời gian cùng với khái niệm chuyển đổi ngữ cảnh.

Mỗi tiến trình được ấn định lượng thời gian rất nhỏ để thực thi. Ở đây, việc chuyển ngữ cảnh diễn ra rất nhanh, tạo ảo giác rằng nhiều tiến trình/tác vụ đang thực thi đồng thời. Đa nhiệm đề cập đến việc có nhiều chương trình, quy trình, tác vụ, luồng chạy cùng một lúc.

(Đa luồng -

Một luồng là đơn vị cơ bản của việc sử dụng CPU. Điều này cho phép các tiến trình được chia thành nhiều luồng có thể chạy đồng thời trong ngữ cảnh của tiến trình. Hãy tưởng tượng, nhiều tab được mở trong trình duyệt của bạn khi các luồng khác nhau chạy trong cùng một quy trình.

Ví dụ- VLC Media Player có thể có 3 luồng. Đầu tiên, để mở trình phát media. Thứ hai, để chơi một bài hát cụ thể. Thứ ba, thêm nhiều bài hát hơn vào danh sách phát.

Giả sử có một trình duyệt web đang xử lý yêu cầu của khách hàng. Trong trường hợp hệ thống đơn luồng, máy chủ web sẽ không thể xử lý các yêu cầu khác cho đến khi xử lý xong yêu cầu hiện tại, điều này thực sự tốn kém và mất thời gian. Như vậy, với khái niệm đa luồng, trình duyệt web khi nhận được yêu cầu mới có thể tạo một luồng mới để xử lý yêu cầu.

Thuận lợi -

Tăng khả năng đáp ứng. Khi một luồng cụ thể kết thúc quá trình thực thi, nó sẽ đưa ra kết quả mà không cần đợi một luồng có thể cần nhiều thời gian hơn để thực thi, tức là nếu một luồng bị chặn thì tất cả các luồng còn lại vẫn tiếp tục phản hồi.

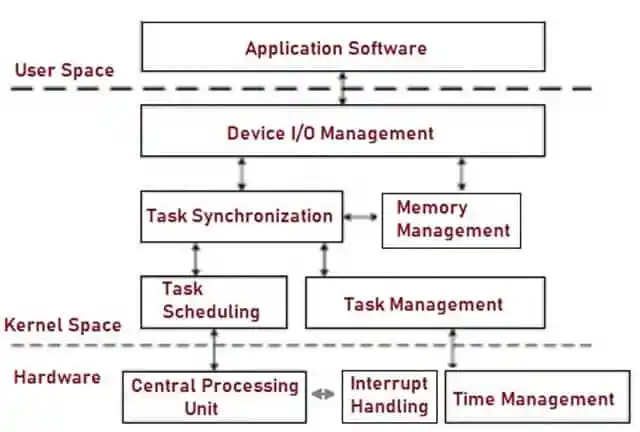
Tạo quy trình mới và phân bổ nguồn lực mới có thể tốn thời gian. Vì các luồng chia sẻ cùng một tài nguyên nên việc tạo một luồng và chuyển đổi giữa các luồng sẽ dễ dàng hơn nhiều.)

# LESSON 4: KERNEL

## KERNLE LÀ LÀ GÌ ?

Kernel - hay được gọi là nhân của hệ điều hành, thực chất là một quy ước có nhiệm vụ điều phối các công vieecjcuar RTOS (gọi là quy ước vì thực tế nó vẫn là do mình code nên và nạp vào bộ nhớ, chứ không phải cái gì đó to tát nhúng từ bên ngoài vào bên trong vi điều khiển của mình). Việc lập trình ra Kernel hay toàn bộ hoạt động của một OS trong vi điều khiển thường được thực hiện bằng ngôn ngữ Assembly, để đảm bảo tốc độ xử lý nhanh nhất của Kernel.

* Kernel sẽ điều phối haotj động của các Task dựa vào bộ lập lịch - Scheduler và các thuật toán lập lịch (do con người quy định).
* Kernel sẽ quản lý tài nguyên phần cứng - bộ nhớ, để lưu trữ hoạt động của các Task.
* Kernel quản lý các công việc giao tiếp giữa các Task, xử lý ngắt…



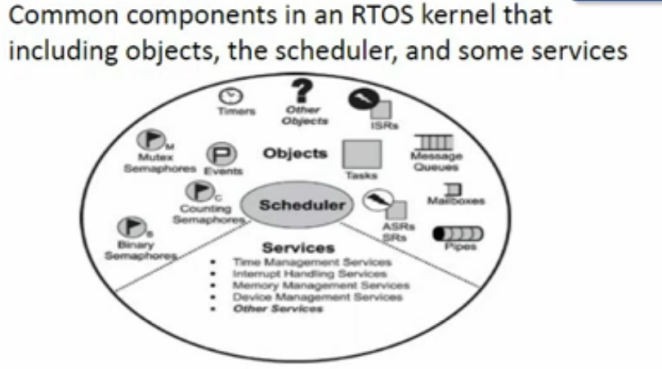
=> Nói chung là mọi công việc của RTOS, Kernel sẽ đại diện để thực hiện mọi công việc.

Manual context switching có thể được tự động hóa bằng phần mềm đặc biệt gọi là Kernel của Hệ Điều Hành Thời Gian Thực, hoặc viết tắt là RTOS-Kernel.

RTOS-Kernel là phần mềm mở rộng kiến trúc nền trước/nền sau cơ bản bằng cách cho phép bạn chạy nhiều vòng lặp nền (gọi là luồng hoặc tác vụ) trên một CPU duy nhất. Công việc chính của RTOS-Kernel là chuyển đổi ngữ cảnh CPU thường xuyên từ luồng này sang luồng khác để tạo ra ảo giác rằng tất cả các luồng này chạy đồng thời. Kỹ thuật này còn được gọi là đa luồng hoặc đa nhiệm.

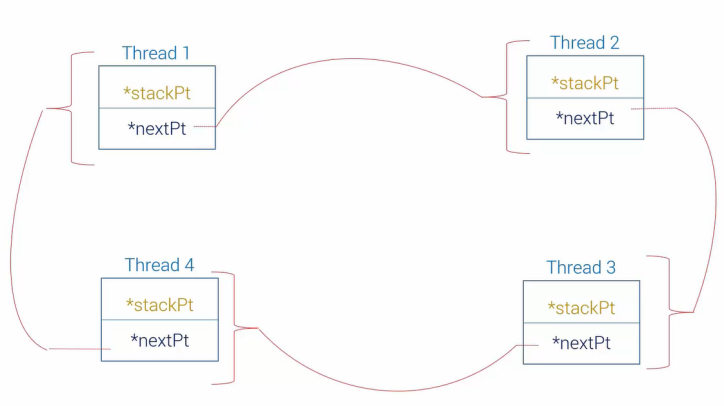
Most RTOS kernels contain the following components:

* Scheduler-is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.
* Objects-are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
* Services-are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

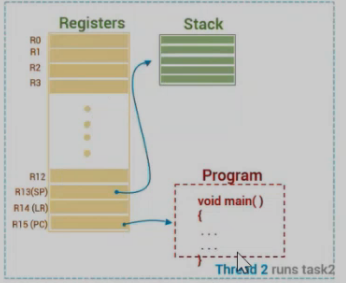
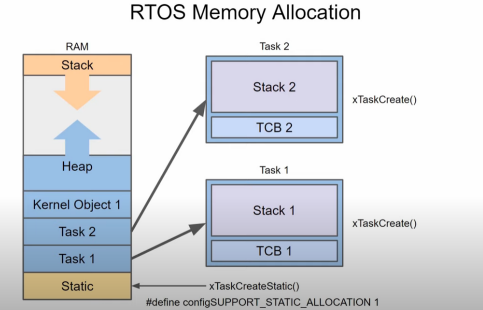


## ROUND ROBIN SCHEDULER DEVELOPING

Theo thuật toán lập lịch RoundRobin, mỗi Task sẽ lần lượt chạy, Kernel sẽ chuyển đến chạy Task tiếp theo khi hết thời gian timeslice (quanta).



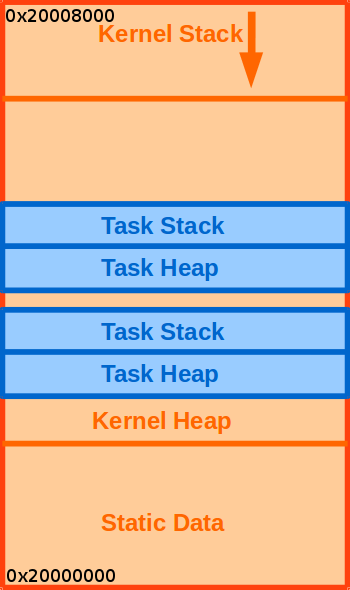
Khi Pointer **currentPt** trỏ đến 1 Thread (TCB), nội dung trong Stack của Thread sẽ được load cho CPU (**16 thành ghi của Register Core**), và Thread đó sẽ được sử dụng CPU (**Context Switching**)

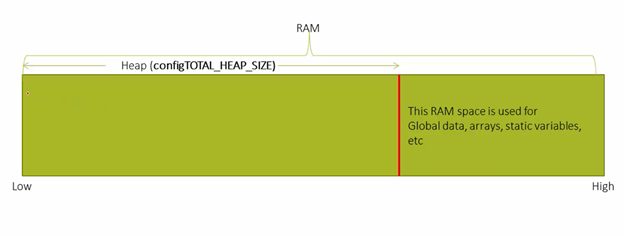


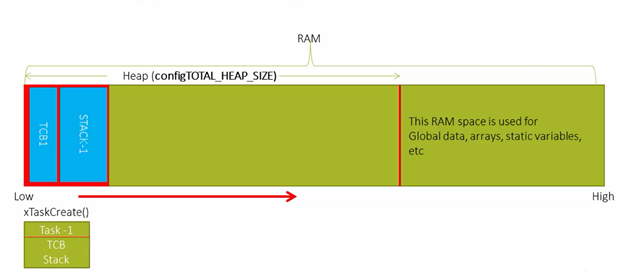
## TCB DEVELOPING

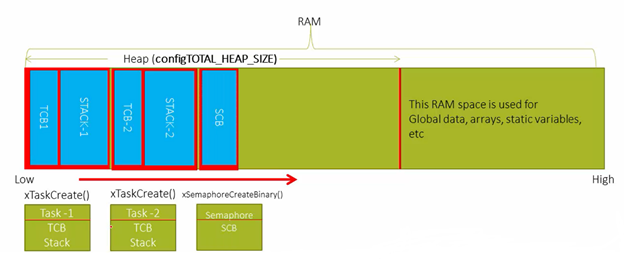
TCB được khai báo như 1 dạng Linked List, chứa thông tin của 1 task (tên task, mức ưu tiên, số thứ tự, con trỏ chỉ tới stack, ...).

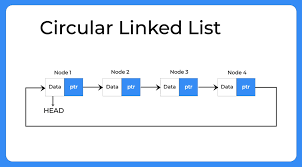
Mỗi task sẽ có 1 stack tương ứng, cần phân biệt với kernel stack (bắt đầu từ vị trí SRAM\_END = 0x20008000). Vậy sẽ có 2 loại stack trong 1 ứng dụng RTOS là task stack (điều khiển bởi PSP) và kernel stack (điều khiển bởi MSP).











BUỔI 4

(https://wiki.csie.ncku.edu.tw/embedded/rtenv)

ARM Cortex-M Exception Handling

(<https://www.youtube.com/watch?v=mNUH0O_4Zn8>)

<https://www.facebook.com/groups/259967441230713/posts/411449969415792/?paipv=0&eav=AfYPvlb9-NMboF5QuQfxYrtn3KSomjY-5Hp1ry9e4MRujUjlL1edVq9p8hO9EIaPRFk&_rdr>

# Lesson 5: Developing Round Robin Scheduler

## Kernel Initialization

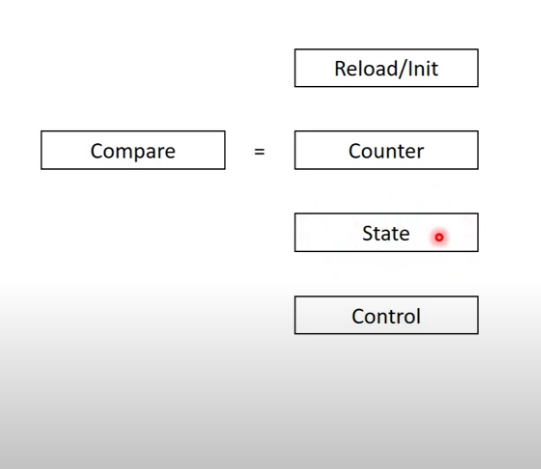
Để Scheduler điều phối đước các Task, cũng như tạo quanta time, Kernel cần có 1 ngắt xảy ra theo chu kì. SysTick Interrupt sử dụng để tạo ngắt cho Kernel.

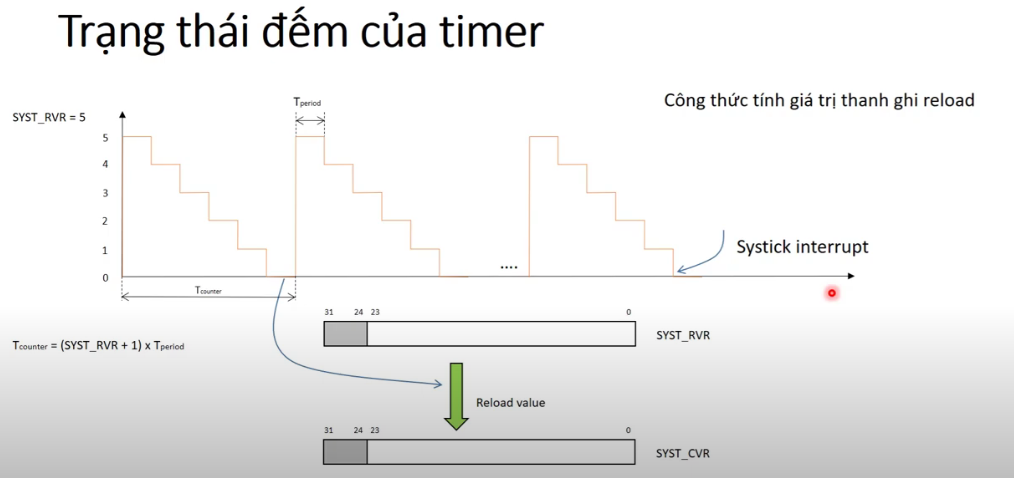
### System Timer (Systick)

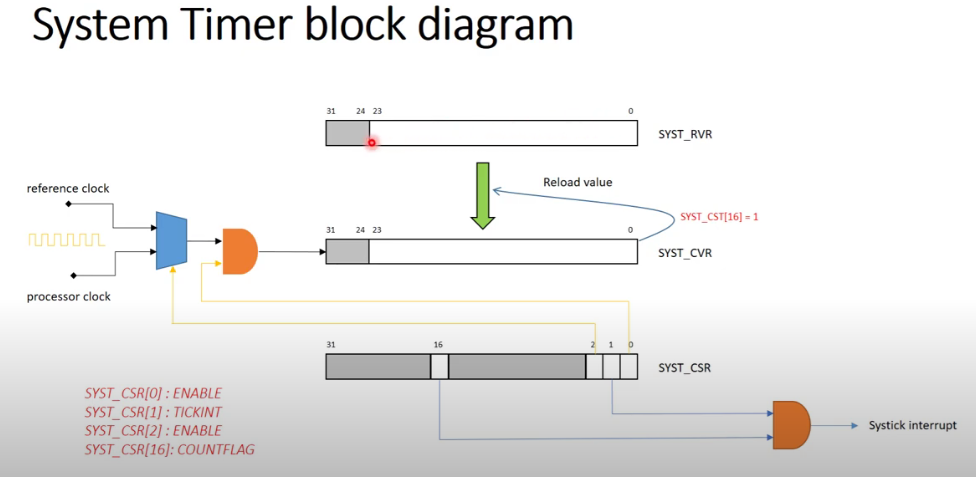
Giới hạn của bộ đếm được xác định dựa trên giới hạn của thanh ghi counter như 8/16/24/32 bit.

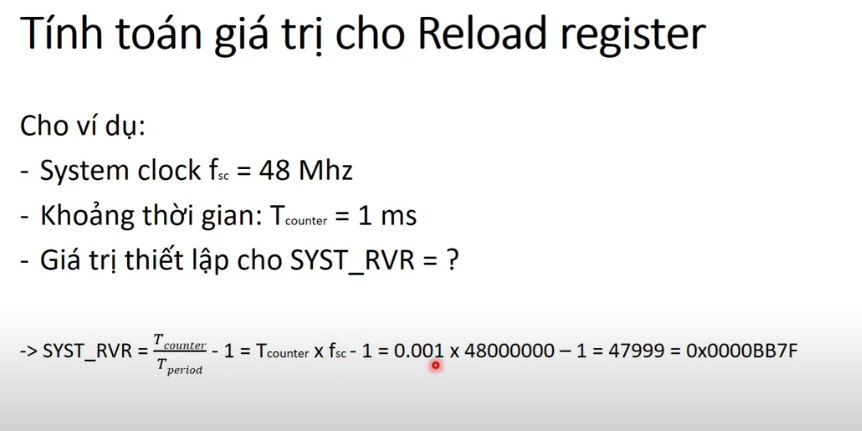
Có 5 nhóm thanh ghi:

* Reload/init
* Compare
* Counter
* State
* Control









Code:

/\*Ex

1s -> Bus\_Freq

1000ms-> Bus\_Freq

1ms -> Bus\_Freq/1000

\*/

void osKernelInit(void){

//1ms

MILLIS\_PRESCALER = (BUS\_FREQ/1000);

}

// Systick

#define \_\_vo volatile

#define SYST\_BASEADDR 0xE000E010

typedef struct {

\_\_vo uint32\_t CSR;

\_\_vo uint32\_t RVR;

\_\_vo uint32\_t CVR;

\_\_vo uint32\_t CALIB;

} STK\_RegDef\_t;

#define SYST ((STK\_RegDef\_t \*)SYST\_BASEADDR)

//Init some value for SysTick Reg

#define CTRL\_ENABLE (1<<0)

#define CTRL\_TICKINT (1<<1)

#define CTRL\_CLCKSRC (1<<2)

#define CTRL\_COUNTFLAG (1<<16)

#define SYSTICK\_RST 0

void osKernelLaunch(uint32\_t quanta){

/\*Reset the SysTick\*/

SysTick->CTRL = SYSTICK\_RST;

// SYST->CSR = SYSTICK\_RST;

/\*Clear Systick current value register\*/

SysTick->VAL = 0;

// SYST->CVR = 0;

/\*Load the quanta\*/

SysTick->LOAD = (quanta\*MILLIS\_PRESCALER-1);

// SYST->RVR = (quanta\*MILLIS\_PRESCALER - 1);

/\*Set Systick interupt to lowest priority\*/

NVIC\_SetPriority(SysTick\_IRQn, 15);

/\*Enable Systick, select Internal clock\*/

SysTick->CTRL |= CTRL\_CLCKSRC|CTRL\_ENABLE;

// SYST->CSR |= CTRL\_CLCKSRC | CTRL\_ENABLE;

/\*Enable Systick interupt\*/

SysTick->CTRL |= CTRL\_TICKINT;

// SYST->CSR |= CTRL\_TICKINT;;

//os launch scheduler

osSchedulerLaunch();

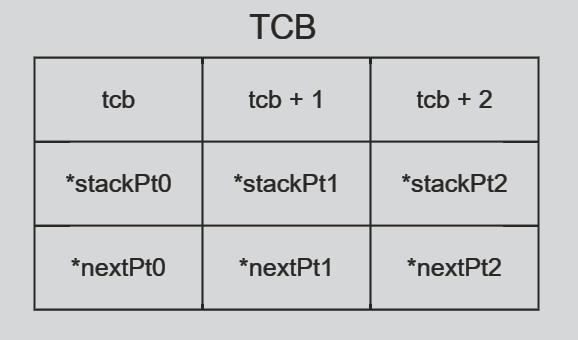
}

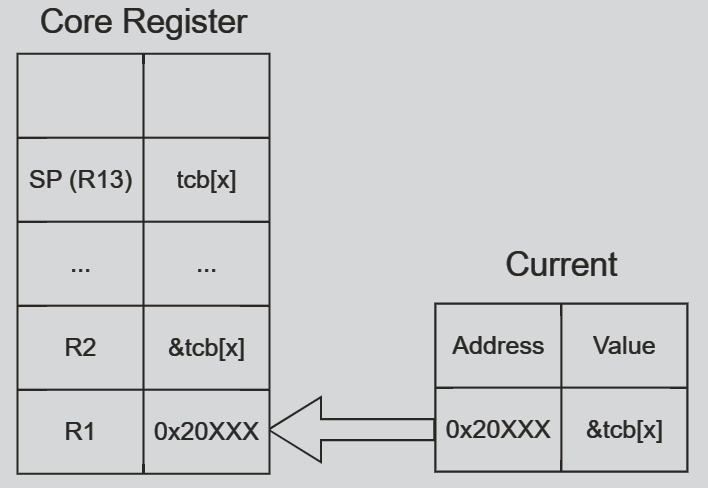
### Launching the Scheduler

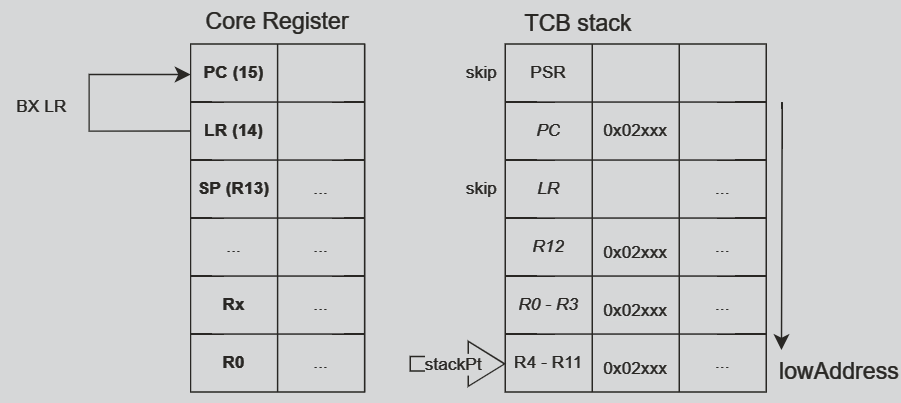
Để bắt đầu hệ thống, Scheduler phải chọn khởi chạy Task đầu tiên bằng hàm osSchedulerLauch().

Các hàm của Scheduler được viết bằng hợp ngữ để đảm bảo tốc độ thực thi nhanh nhất.

* Xác định Task chạy đầu tiên bằng currentPt
* Địa chỉ của TCB chính là địa chỉ của stackPt của TCB đó.
* Để khởi chạy, cần load các thanh ghi trong Stack cho CPU.
  + POP các thanh ghi theo thứ tự.
  + Vì là Task đầu tiên chạy, bỏ qua LR.
  + Nạp LR bằng PC trong Stack
  + Bỏ qua PSR
* Khởi chạy bằng cách cho hàm osSchedulerLaunch return.
* Khi hàm trả về, nó nạp PC bằng giá trị LR để xác định vị trí tiếp tục của chương trình.





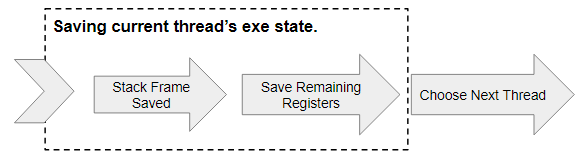


### SysTick Handler

SysTick sẽ đảm nhận việc định thời quanta time cho Scheduler.

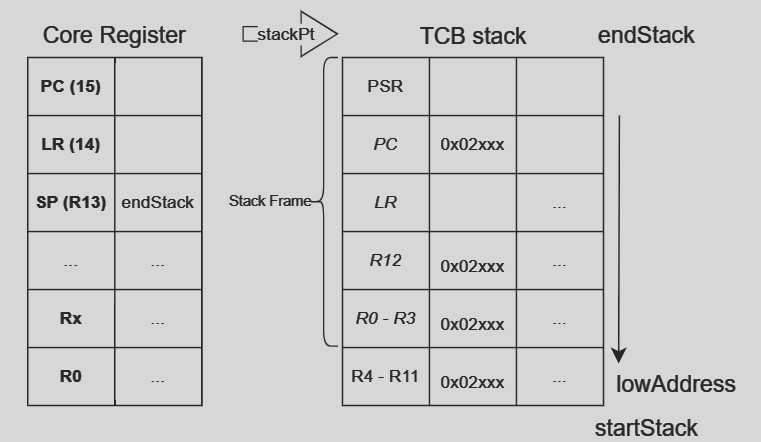
Sau mỗi quanta, SysTick sẽ tạo ngắt và gọi hàm xử lý ngắt SysTick Handler.

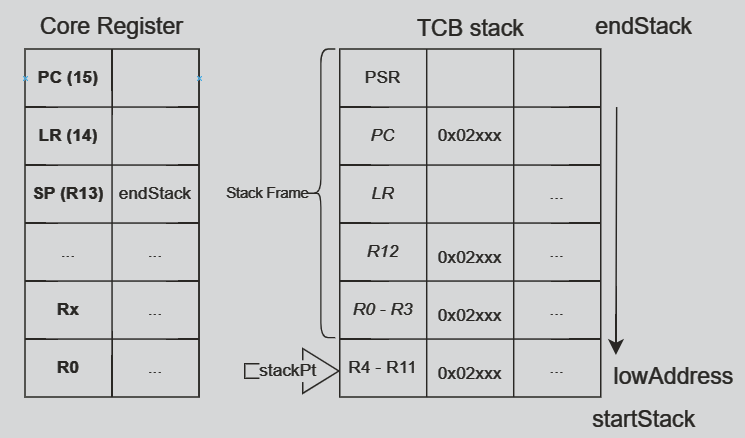
Quá trình Switching Threads sẽ diễn ra trong exception này.



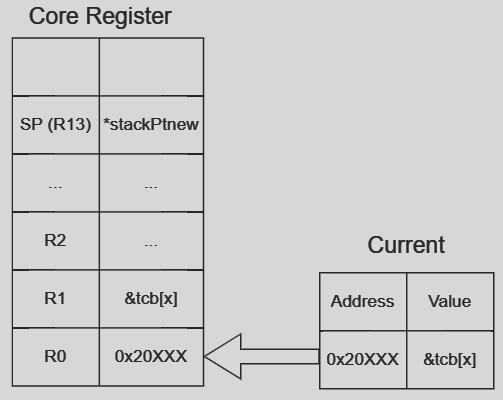
Quá trình là exception nên Stack Frame sẽ được lưu tự động.

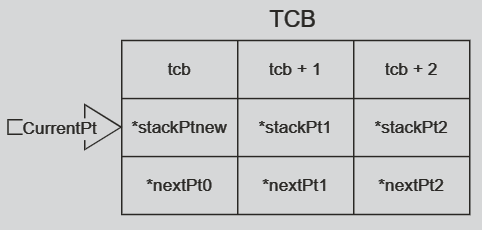
Các thanh ghi còn lại được lưu vào Stack của Thread.

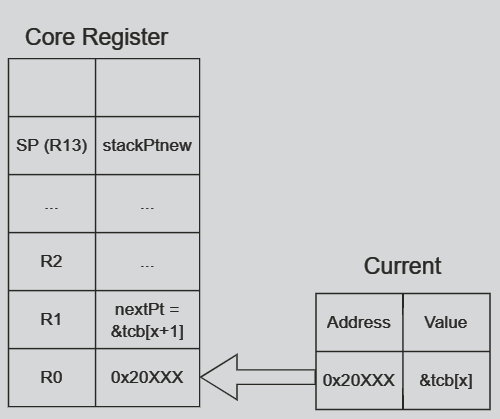




* Stack Frame được lưu tự động.
* Các thanh ghi còn lại được lưu vào Stack của Thread.

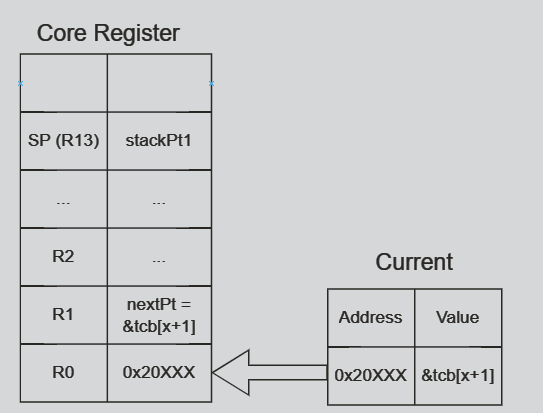


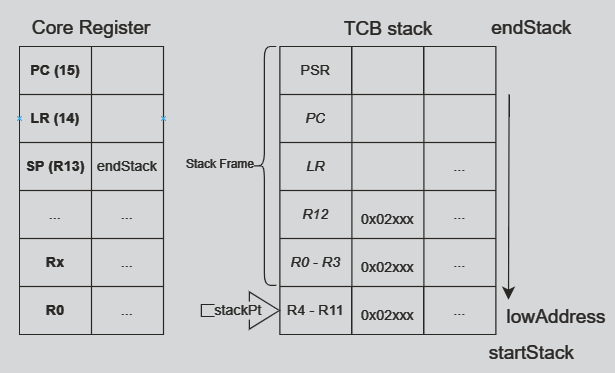


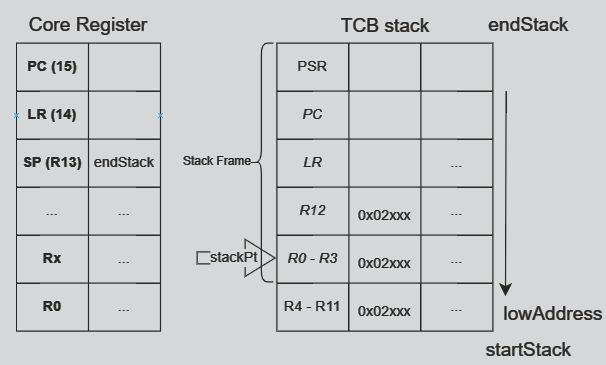


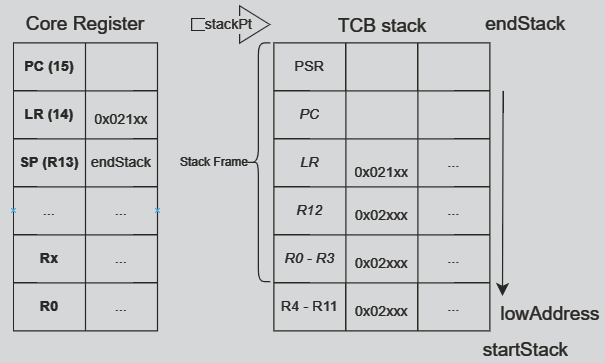


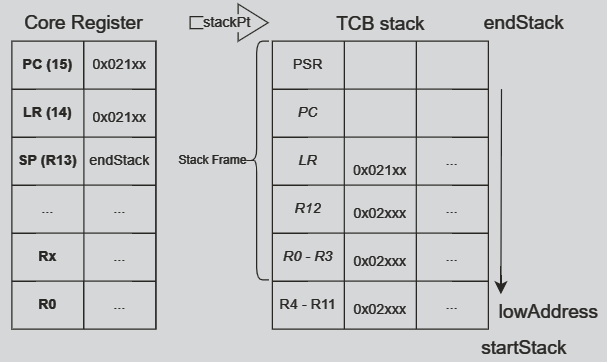


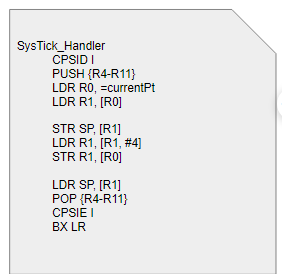












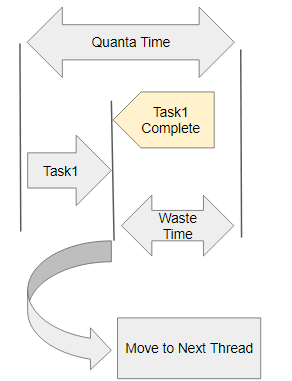
# Lesson 6: Semaphore

## Testing the RTOS

Nếu 1 Task hoàn thành công việc của nó trước quanta time, khoảng thời gian còn lại nó phải đợi.

=> lãng phí thời gian.

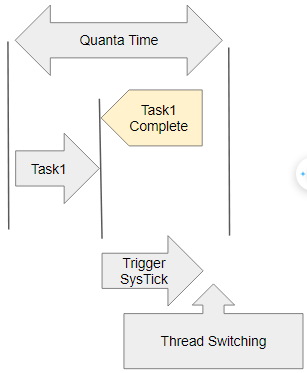
=> Ngay khi 1 Task hoàn tất, lập tức chuyển đến Thread tiếp theo mà không cần đợi hết quanta.



## Thread Yield

Hàm osThreadYield() giúp chuyển đổi Thread mà không cần kết thúc quanta.

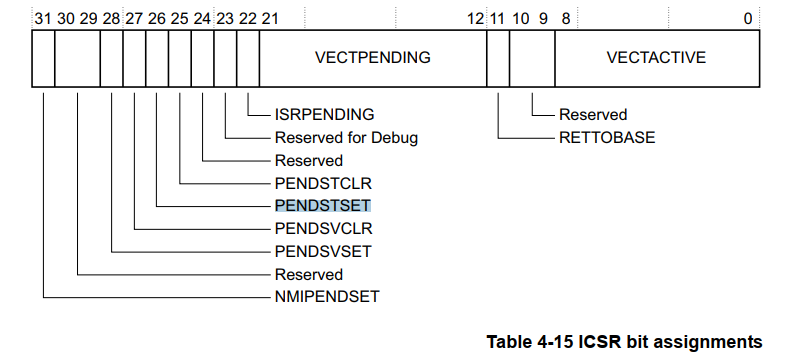
* Hàm này chủ động kích hoạt SysTick.
* Khi SysTick xảy ra, Thread sẽ tự động được chuyển.
* Tạo tín hiệu kích hoạt SysTick Handler



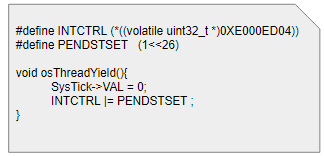
## Cooperative Scheduler

Một Task có thể chủ động gọi hàm Yield để nhường cho Task tiếp theo mà không cần chờ hết quanta.

=> Giúp tối ưu hóa năng suất của hệ thống.



* Bit số 26 của thanh ghi ICSR (PENDSTSET) diều khiển trạng thái ngắt SysTick.
* Ngắt SysTick sẽ được kích hoạt khi Bit này được set lên 1.

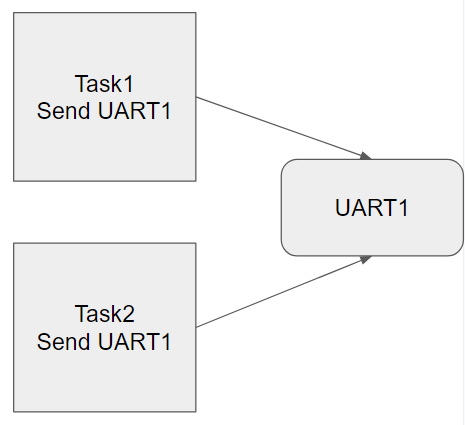
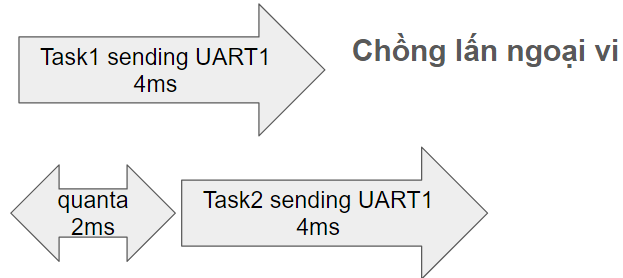


## Semaphore

### Testing the RTOS

Khi nhiều tác vụ cùng truy cập để sử dụng 1 tài nguyên (ngoại vi, biến, vùng nhớ, …), hệ thống sẽ không đủ tài nguyên để chạy (time quanta, Stack, …).

Việc không đủ Stack có thể gây Crash hoặc quanta time không đủ có thể gây chồng lấn trên cùng 1 ngoại vi.

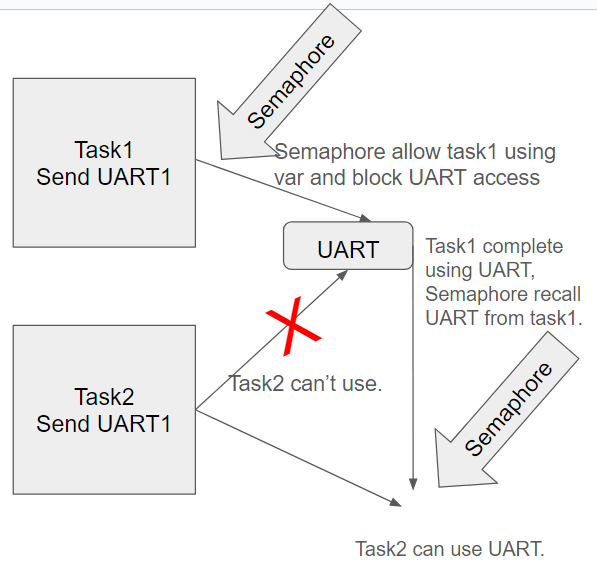
 

### Semaphore

Trong RTOS, Semaphore là 1 ứng dụng có nhiệm vụ phân phát và thu hồi tài nguyên.

* Khi 1 Task muốn sử dụng 1 tài nguyên, nó phải chờ Semaphore cấp quyền.
* Khi Task sử dụng xong, Semaphore sẽ thu hồi lại tài nguyên này.
* Tại 1 thời điểm, chỉ có 1 Task được cấp quyền.

=> Ngăn chặn được việc nhiều Task cùng sử dụng 1 tài nguyên.



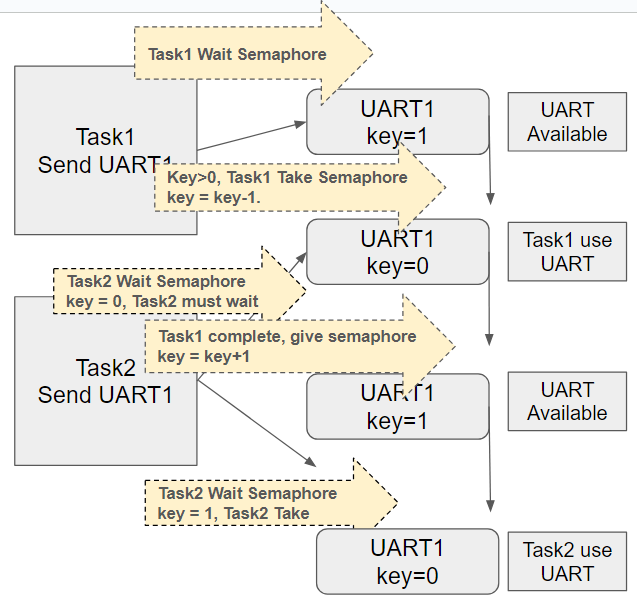
### Hoạt động

Semaphore sử dụng 1 key (hay flag) để báo trạng thái của 1 tài nguyên:

* Key = 0: tài nguyên đang được sử dụng.
* Key > 0: tài nguyên khả dụng.

Khi 1 Task sử dụng tài nguyên, nó giảm Key đi 1 đơn vị - Take Semaphore.

Khi 1 Task muốn sử dụng tài nguyên, nó phải đợi để được Take Semaphore - Wait Semaphore.



### Các loại Semaphore

#### Binary Semaphore

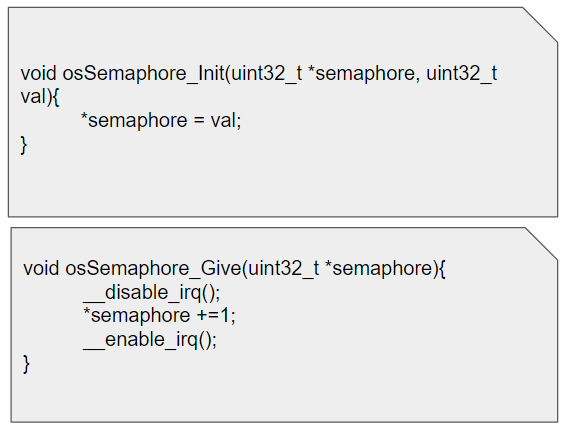
* Key chỉ có 2 giá trị 0 và 1.
* Đồng bộ 1 hoạt động duy nhất.
* Hoạt động tương tự 1 Queue có độ dài 1.

#### Counting Semaphore

* Key có giá trị >= 0.
* Dùng đếm sự kiện:
  + Khi có sự kiện xảy ra thì semaphore +1.
  + Mỗi Task xử lý 1 sự kiện sẽ trừ semaphore đi 1.
* Hoạt động tương tự 1 Queue có độ dài lớn hơn 1.

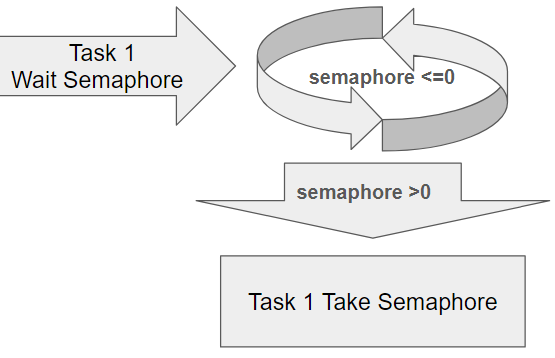
### Developing Semaphore

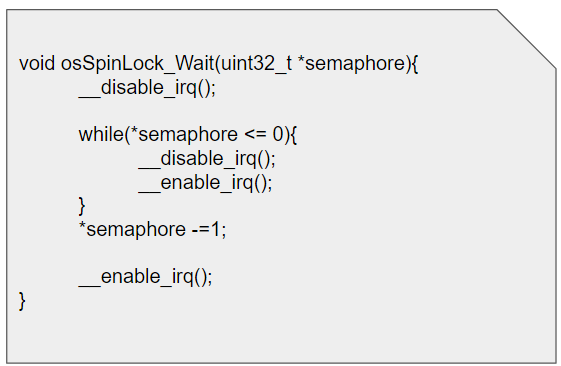
* Hàm osSemaphore\_Init set giá trị ban đầu cho các semaphore.
* Hàm osSemaphore\_Give thực hiện give semaphore cho 1 Task.



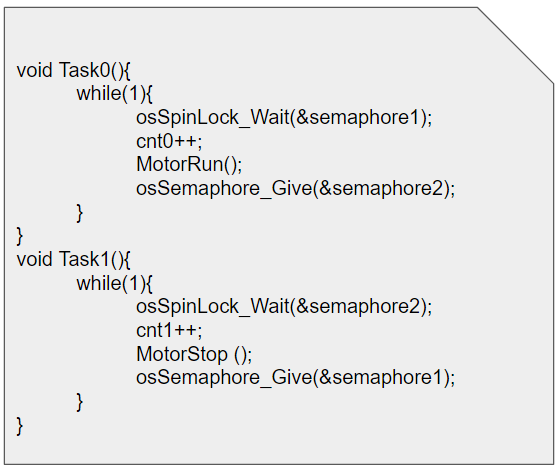
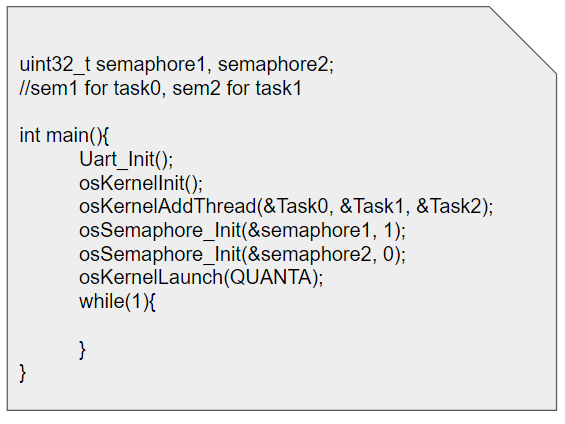
### SpinLock Semaphore

Khi 1 Task wait 1 semaphore, nếu semaphore đó không khả dụng, hàm SpinLock sẽ chờ semaphore trong 1 vòng lặp.



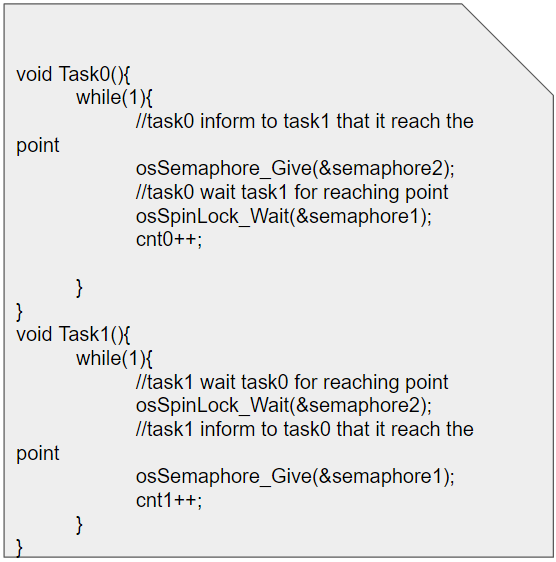


### Synchronizing Task using Semaphore



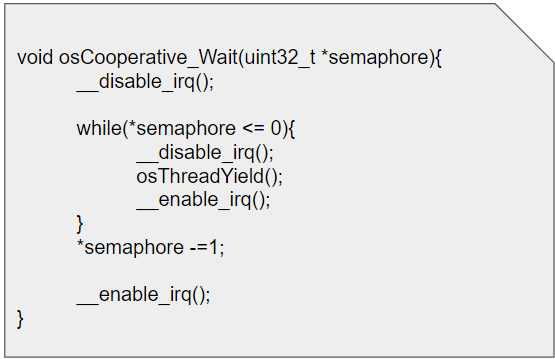
### Rendezvous

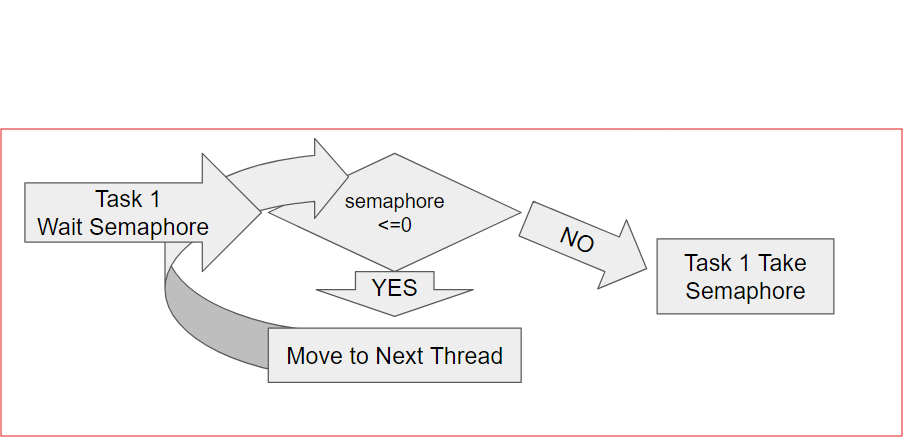
* VD: ta muốn tại 1 vị trí cụ thể trong chương trình (điểm hẹn cụ thể ), Task 1 và 2 sẽ cùng chạy.
  + Khi 1 Task đến điểm hẹn, nó phải đợi Task kia để cùng tiếp tục chạy.
  + Semaphore có thể được sử dụng để chờ các Task chỉ định đến điểm hẹn.



### Cooperative Semaphore

* Việc sử dụng vòng lặp như ở SpinLock gây lãng phí thời gian.
* Cooperative Semaphore thực hiện việc chuyển Task trong lúc đợi Semaphore.





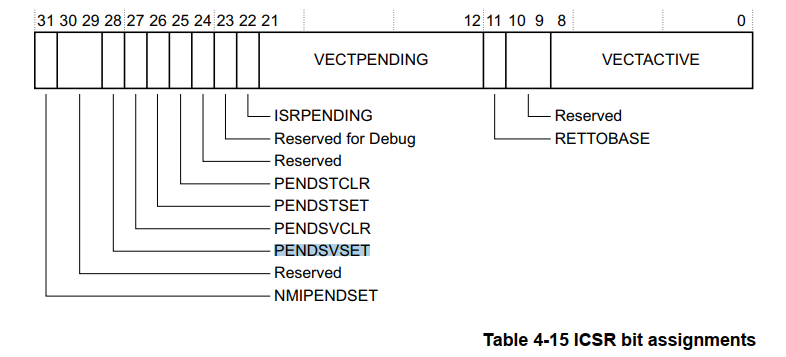
# Lesson 7: Periodic Scheduler

## PenSV

* PenSV là 1 tính năng cho phép kích hoạt exception tương tự ISR.
* Khác với các ISR, thường là riêng biệt đối với từng loại thiết bị, PendSV là một phần của Cortex-M nên được sử dụng chung trên tất cả các dòng vi điều khiển Cortex-M.

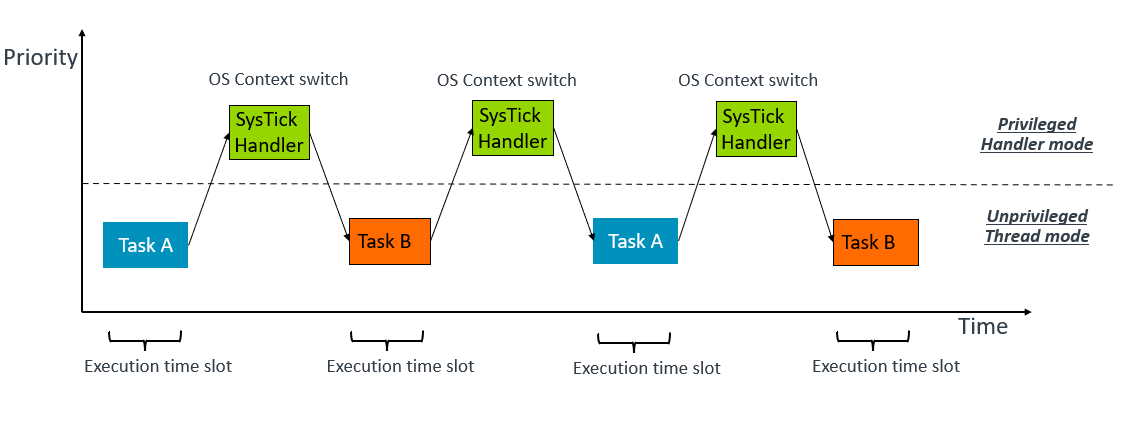
=> Khi phát triển các OS, PendSV được sử dụng chung để không cần phải tùy chỉnh cho từng dòng VĐK.

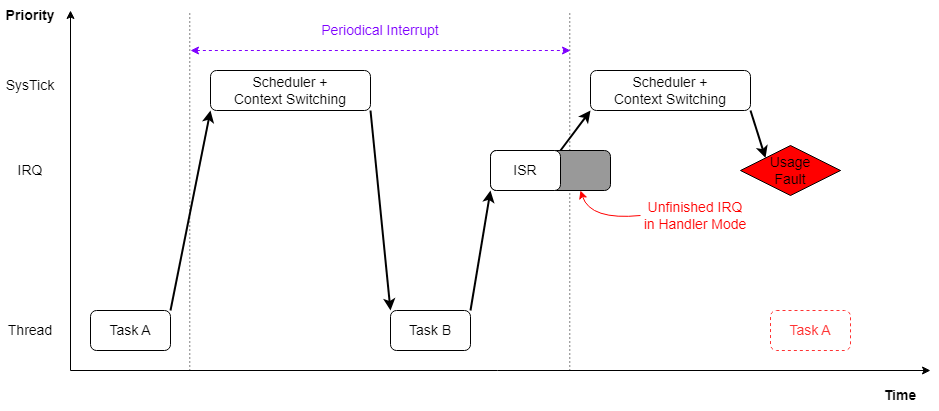
* PenSV được kích hoạt bằng cách Set bit PENDSVSET trong thanh ghi ICSR.
* Mặc định PendSV có ưu tiên thấp nhất.



Ở bài trước, chung ta thiết kế OS sử dụng SysTick Handler để tạo exception thực hiện Context Switching.

Tuy nhiên nếu SysTick ISR có độ ưu tiên cao và đang có 1 ISR khác xảy ra, quá trình Context Switching có thể gây gián đoạn ISR.

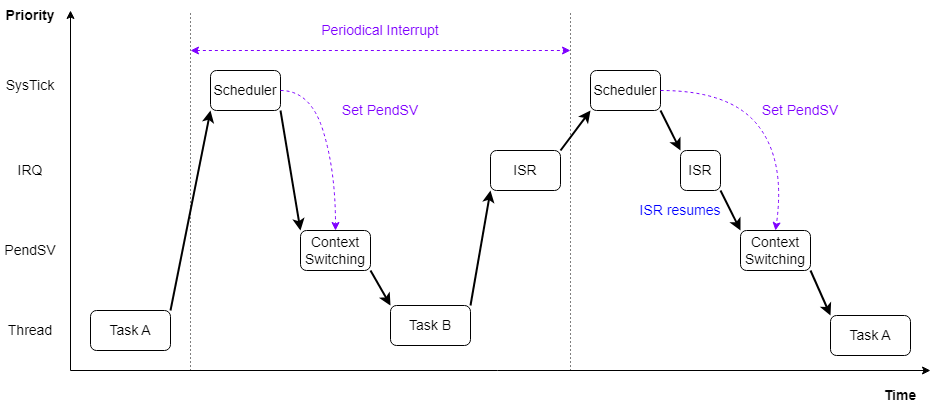




Sử dụng PendSV để thực hiện Context Switching sẽ tránh được các tính huống làm gián đoạn khi đang thực thi ISR.

=> Tách việc Context Switching khỏi SysTick Handler bằng cách đặt nó vào PendSV Handler.

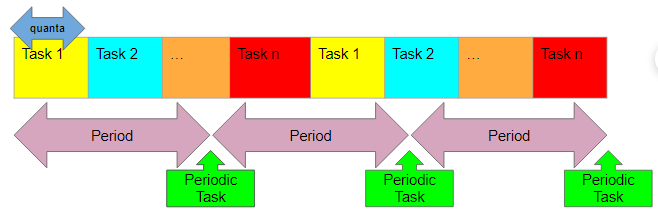
=> Context Switching sẽ thực hiện khi không còn ngắt nào đang chờ.



## Periodic Task

Periodic Task là những tác vụ chạy sau mỗi khoảng thời gian nhất định.

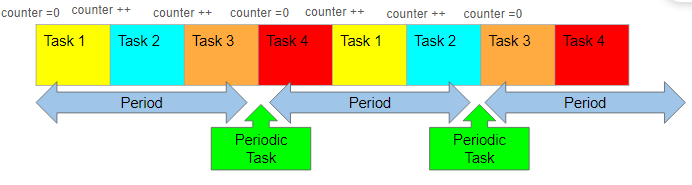
Thường được dùng để xử lý các tác vụ một cách định kỳ.



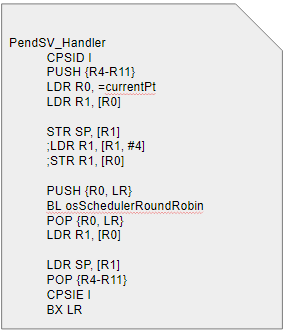
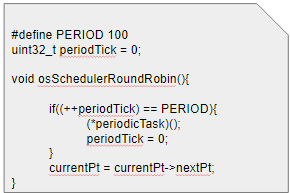
### Periodic Scheduler using Counting Variable

Có thể dùng quanta time để định chu kỳ của Period Task.

Mỗi lần thực hiện Switch Task, cập nhật biến đếm để xác định thời điểm chạy Periodic Task.



* Thay thế việc cập nhật currentPt sau mỗi quanta bằng hàm osSchedulerRoundRobin.
* Trong hàm này, kiểm tra biến đếm để xác định thời gian chạy của Periodic Task.



### Multi-Periodic Task using Counting Variable

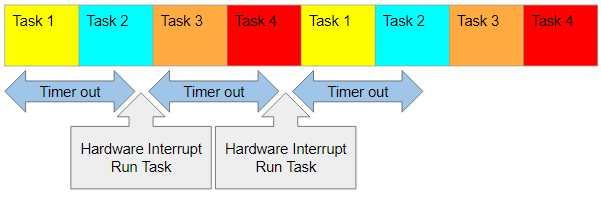
Bằng cách thay đổi các điều kiện khi cập nhật biến đếm, có thể dễ dàng khởi chạy các Task với các chu kì khác nhau.



### Periodic Scheduler using Hardware Interrupts

Các ngoại vi như Timer có thể tạo ra các ngắt theo chu kỳ định sẵn.

Sử dụng các ngắt này, chỉ cần gọi chương trình của Periodic Task trong chương trình xử lý ngắt.

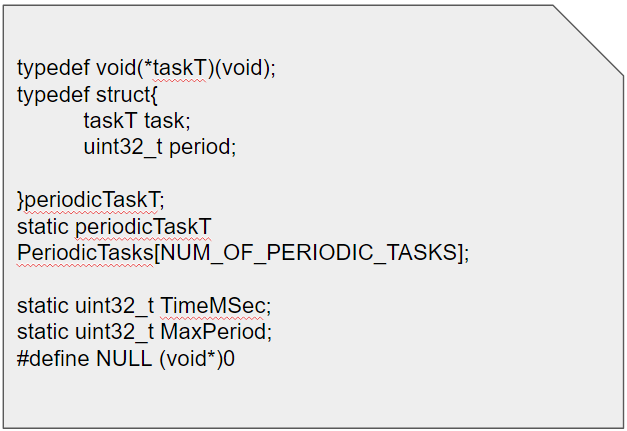


### Periodic Scheduler using TCB

Tương tự các Task thông thường, TCB cũng có thể dùng để quản lý các Periodic Task.

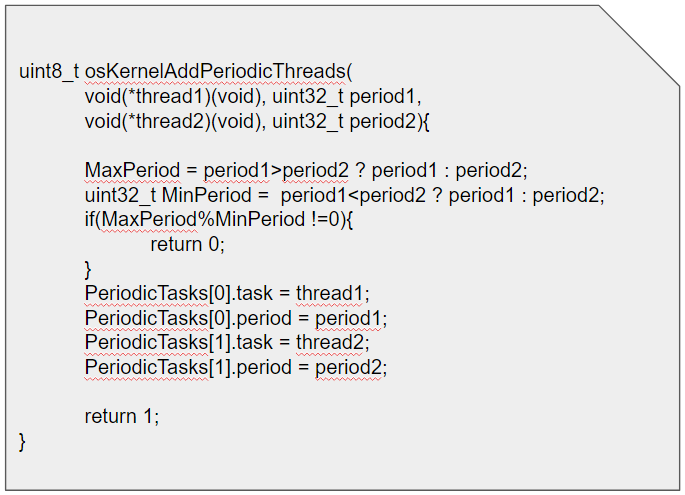
Gồm:

* Con trỏ đến Periodic Task.
* Chu kì của Task đó.



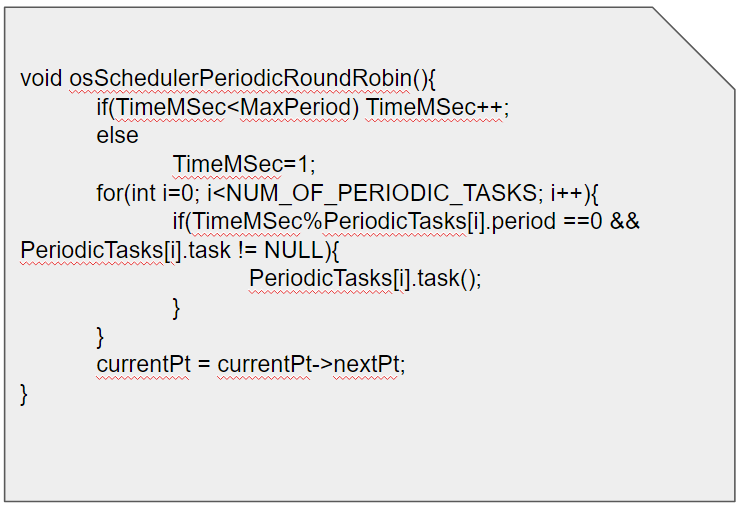
#### Add Periodic Tasks to OS

* Các periodic task được quy định với chu kỳ là bội số của nhau để dễ quản lý.
* Hàm osKernelAddPeriodicThread thêm các Task vào OS.



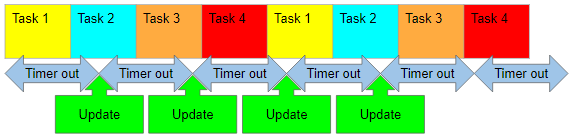
#### Run Periodic Tasks

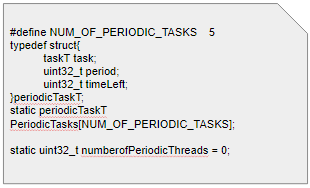
* Mỗi lần hết quanta, cập nhật biến đếm và kiểm tra để chạy các Task nếu đã đến chu kỳ.
* Sau khi các kiểm tra xong, tiến hành chuyển đổi Task chính như thường lệ.



#### Periodic Scheduler using TCB & Hardware

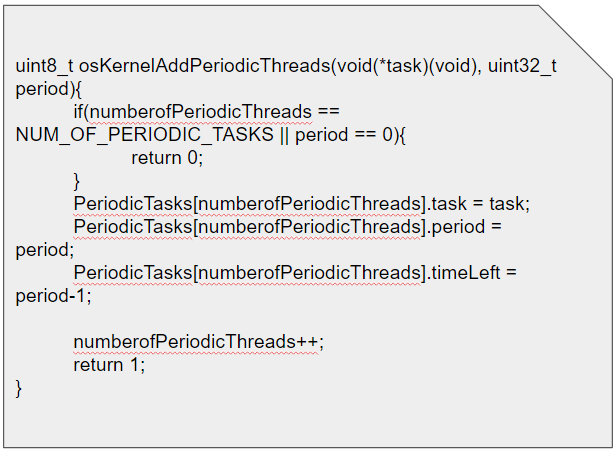
* Giá trị timeLeft dùng để theo dõi Period của Task. Nếu biến này =0, Task đến chu kỳ cần thực thi.
* timeLeft sẽ được cập nhật mỗi khoảng thời gian nhất định, tạo bởi Harware ( Timer).

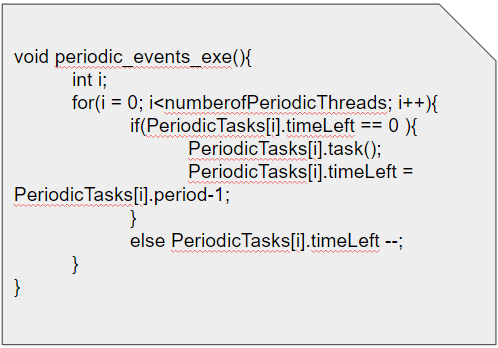




### Add and Update the Task

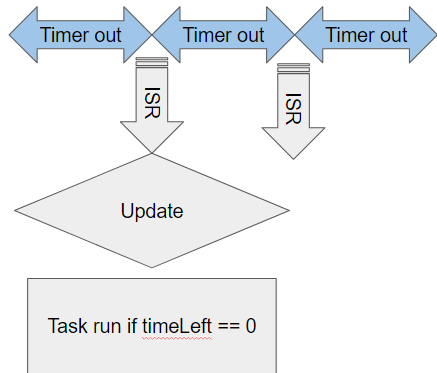
* Hàm osKernelAddPeriodicThreads thêm các Periodic Task vào OS với chu kỳ nhất định.
* Cập nhật và khởi chạy các Task khi đến chu kỳ.

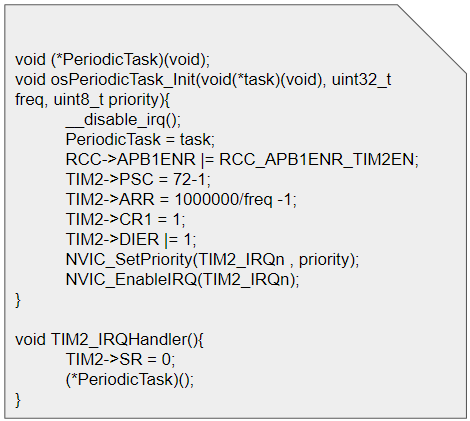




### Init Harware

Ngắt Timer được dùng để tạo các thời điểm cập nhật.

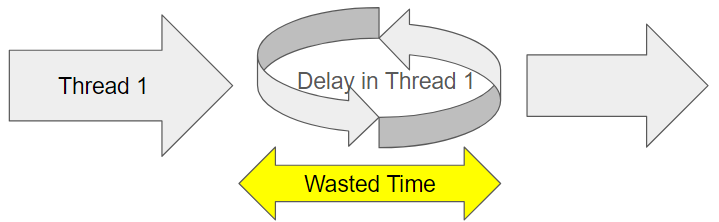




# Lesson 8: Sporadic Scheduler

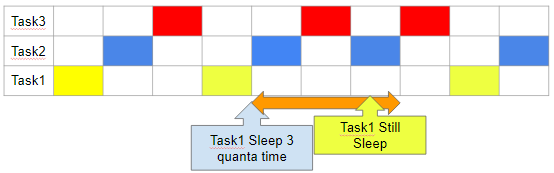
## Sleep Threads

* Khi muốn 1 Thread tạm ngừng trong 1 khoảng thời gian cụ thể, có thể sử dụng delay.
* Tuy nhiên hàm delay thông thường sẽ gây lãng phí tài nguyên.



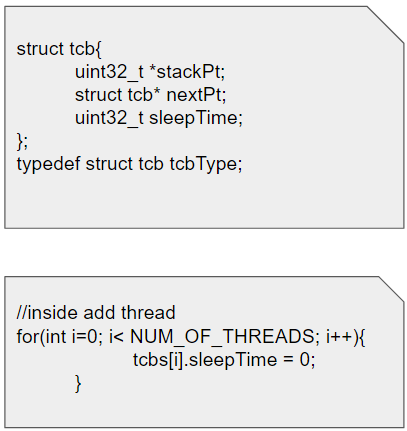
=> Việc đưa Thread vào trạng thái Sleep cho phép 1 Thread tạm ngừng nhưng không ảnh hưởng đến các Thread khác.

* Hàm osThreadSleep() cho phép 1 Thread dừng trong 1 khoảng thời gian.
* Khi 1 Thread ở trạng thái Sleep, Scheduler sẽ bỏ qua và Switching đến Thread tiếp theo.

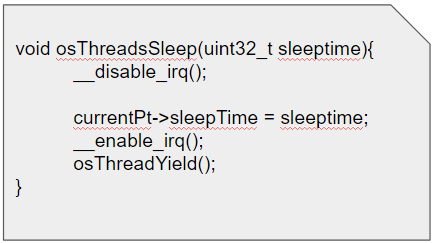
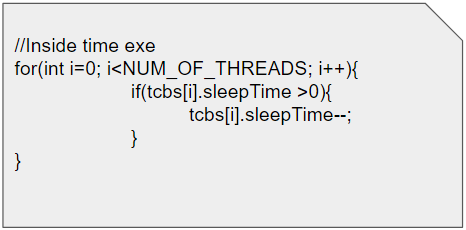


## Develop Thread Sleep API

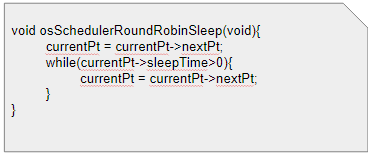
* SleepTime được thêm vào TCB để theo dõi thời gian của các Thread.
* Ở hàm add thread vào OS, khởi tạo sleepTime cho các Thread ban đầu =0.



* Sleeptime sẽ được cập nhật liên tục sau mỗi khoảng thời gian nhất định. Để không bị giới hạn bởi quanta time, có thể dùng timer.
* Hàm osThreadSleep() khi gọi sẽ đưa thread vào trạng thái sleep.
  + Set giá trị cho sleepTime của thread.
  + Lập tức chuyển đến Thread tiếp theo.

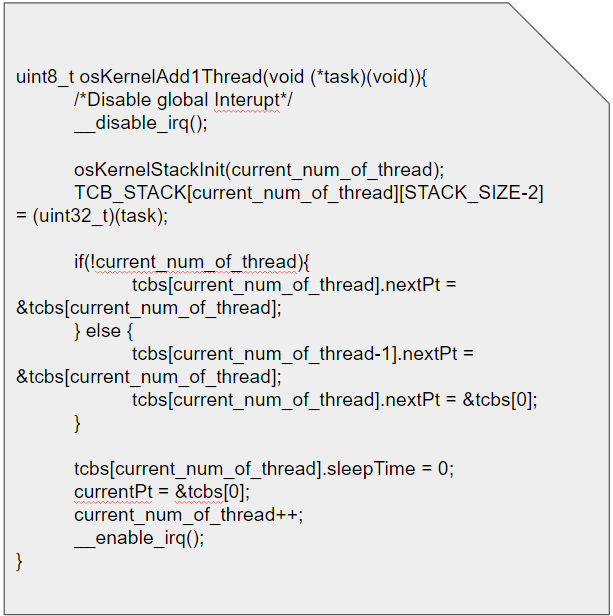


### Scheduler with Sleep



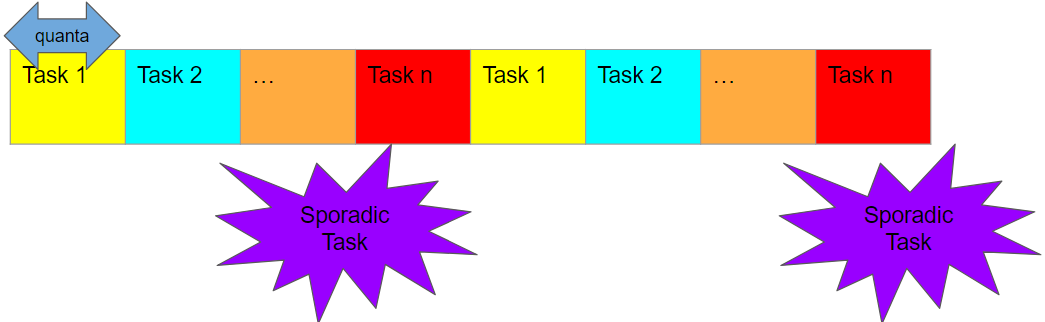
### Adding 1 Threads

Thay thế cho hàm AddThread cũ, có thể thêm lần lượt các Task với số lượng tùy ý.

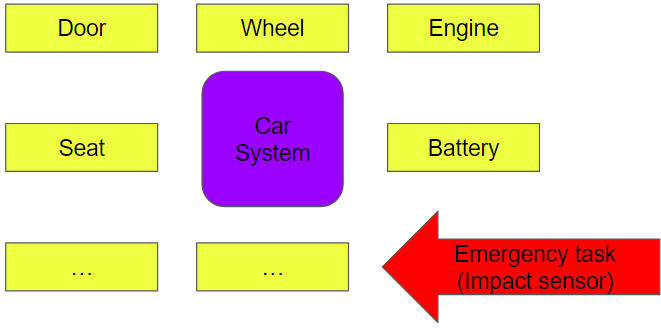


## Sporadic Tasks

Sporadic Task là những tác vụ chạy không thể đoán trước được thời điểm. Nó có thể được khởi chạy tại bất cứ thời điểm nào trong hệ thống.



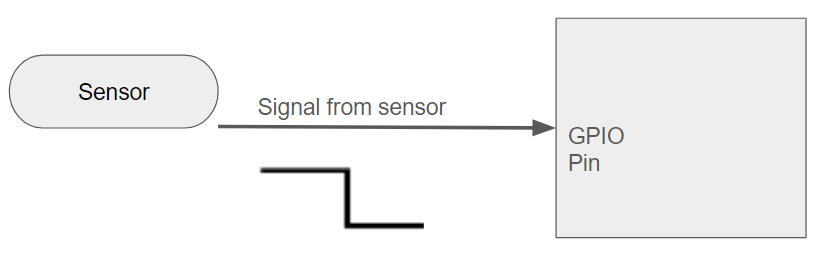
Thường được thiết kế để phục vụ nhưng tình huống quan trọng có thể xảy ra trong hệ thống.



### Sporadic Tasks Example

Ví dụ: Hệ thống có 1 cảm biến nguy hiểm, cảm biến này sẽ thay đổi điện áp trên 1 pin của VDK nếu phát hiện nguy hiểm.

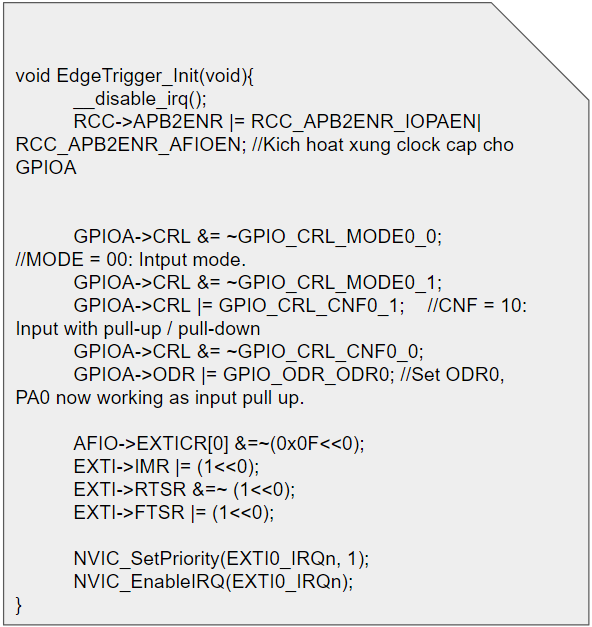
1 Sporadic Task sẽ được cấu hình để phục vụ xử lý nguy hiểm trên.



=> Có thể sử dụng ngắt ngoài để theo dõi tín hiệu.

Cấu hình Pin:

* Cấu hình chân là Input Pull-Up
* Cấu hình Ngắt ngoài để theo dõi tín hiệu ở Sensor.
  + Ngắt ngoài tương ứng với chân PA0.
  + Tín hiệu kích hoạt là Falling edge.



### Launching Sporadic Task

Sử dụng Semaphore để khởi chạy 1 tác vụ khẩn cấp.



