PAGE
DATE

Lec 1: "RTOS"

⇒ what is RTOS ?

Real Time operating system, it's an operating sys designed for systems where the correctness of the sys's operations depends on timely and predictable execution of Tasks or processes. unlike general purpose operating sys "windows, Linux ..." which are optimized for flexibility and multitasking, RTOS is specifically designed to provide deterministic and predictable response times for various taskes.

⇒ what is Determinism ?

Knowing what happens at every point in time line

↵ بقدر أعرف أي اللي يحصل عند نقطة/زمن معين في الـ timeline

⇒ responsiveness ?

How fast I can respond to external events

↵ هل بقدر أستجيب للإدخالات الخارجية بسرعة ولا لأ ؟
وبالتالي حسب إستجابتي هيتحدد أنا كـ sys الـ responsiveness بتاعتي عالية ولا لأ ؟

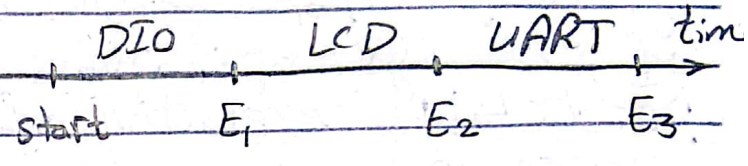
Ex: ترار خارجي ابتدى وبناءً عليه كان المفروض فيه Action هيتنفذ فهل الـ Action ده أتنفذ بسرعة بعد ما الترار ابتدى ولا أخد وقت كبير ده يرجع لقد أي external events الـ sys ده respond

PAGE
DATE

عندنا مجموعة من ال Sys عايزين نقارن ال determinism و ال responsiveness بينها:

① Supper loop sys:

```
while (1) {
    E1 = DIO_set();
    E2 = LCD();
    E3 = UART();
}
```

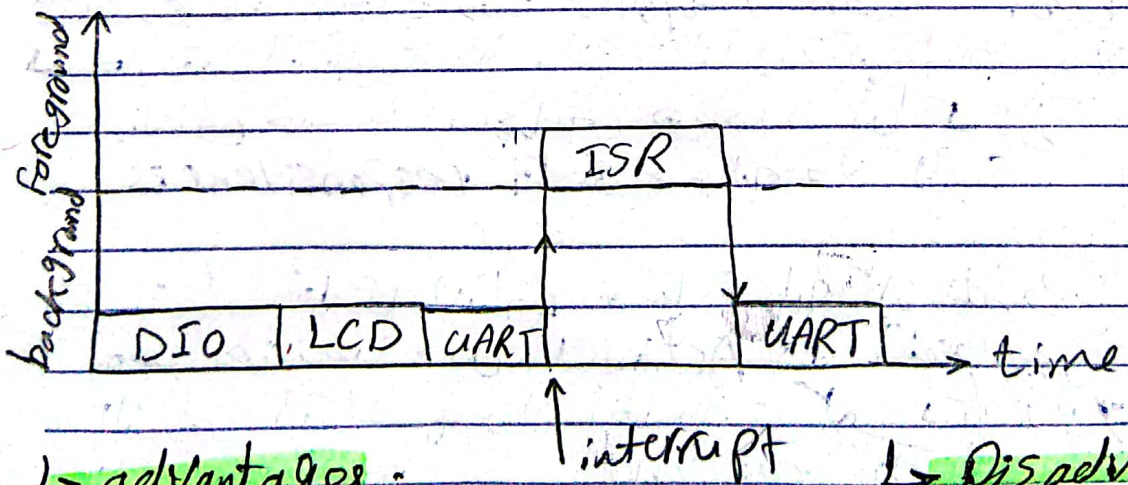


↳ advantages:
1- High determinism
أقدر فأي وقت أعرف ال Sys بينفذ فأي وقت.
2- Simple
3- Minimal Hardware resources

↳ Disadvantages:
1- Low responsiveness
الاستجابة بطيئة جداً للأحداث الخارجية interrupts مثلاً
2- high Power Consumption

② Foreground / background Sys:

ISR ↲ "interrupt" ↳ main ال "super loop"



↳ advantages:
1- High responsiveness

↳ Disadvantages:
1- Low determinism
2- extera HW resource
3- Complex SW

PAGE
DATE

③ RTOS :

الـ RTOS هـ جمع المميزات من الـ 2 sys فهو يتميز بـ :

1- High determinism
2- High responsivness

High determinism
عن طريق استخدام فكرة الـ periodicity

High responsivness
عن طريق استخدام الـ interrupts .

↳ operating sys (O.S) :
sys that tacks a number of tasks and execute them every specific time called periodicity

↳ periodicity → يعني الـ Task ده هيتكرر كل قد أي

↳ Real time :
Correct function at correct time .

Real time types

Hard	firm	Soft
عبارة عن sys لو task ماتفذتش الـ في الزمن المحدد هتسبب human loss في EX: air bag - defence sys ↳ tolerance < 10% ↳ most expensive	لو ماتفذتش بالـ deadline هتسبب خسائر مادية كبيرة جداً EX: - البورصة ↳ Tolerance 10% < T < 40%	لو ماتفذتش بالـ deadline مش هتسبب في أي human loss ولا فقدان أموال كبيرة EX: - Video games ↳ Tolerance > 40% ↳ least expensive

1→ RTOS Composition :

① Task :

→ Job that has to be done independently of the other tasks of the sys.

→ every Task thinks that it has the cpu all to itself.

ال Task عبارة عن ← C function + timing characteristics + Context + storage

1- timing characteristics :

تكون عبارة عن أسئلة متعلقة بال time زي :

1- peridicity
2- execution time
3- deadline
4- Max blockage time

2- Context : "cpu state"

لما بنقل من Task لأخرى لازم أعمل save لبعض الحاجات زي ←

→ PC Program Counter
→ PSW
→ SP stack Pointer
→ cpu register

3- storage :

كل Task بيكون ليها ال TCB الحاجات اللي بيتم عملها save ال data بتاعتها

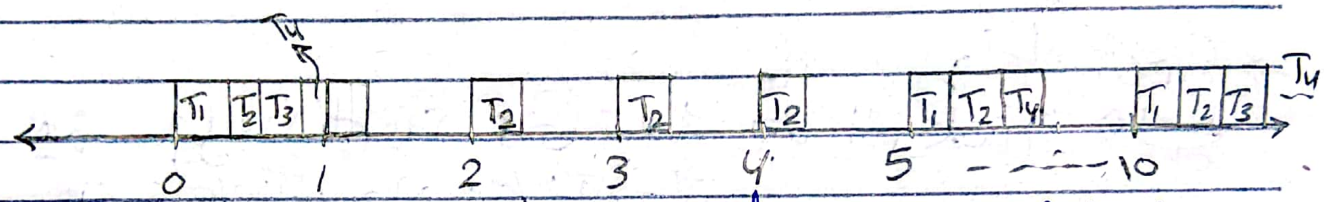
TCB ↓ "Task Control Block"

↳ Cpu Load :

هي أكثر فترة كان الـ Sys مشغول فيها بتنفيذ Tasks وكان
بنسبة قد أيه ؟

EX: we have 4 Tasks , Sys tick = 1 ms

Task	periodicity	execution time
T_1	5 ms	600 μs
T_2	1 ms	100 μs
T_3	10 ms	400 μs
T_4	5 ms	500 μs



Cpu load at tick ⓪ = $\frac{\text{execution time of tasks}}{\text{Sys Tick}}$

$= \frac{600 + 100 + 400 + 500}{1000} = 160\%$

Cpu load at tick ⑤ = $\frac{T_1 + T_2 + T_4}{1000} = \frac{600 + 100 + 500}{1000}$
$= 120\%$

Cpu load at tick ⑩ = $\frac{600 + 400 + 100 + 500}{10000} = 160\%$

↲ فبالتالي الـ Cpu عند أسوء tick خلي الـ Sys بيـ 160%
ولكن يفضل إنها لا تزيد عن 70% أو 80% عشان ؛
لا تكون الـ Sys ← Loaded

تحسين

↳ How to enhance Cpu load ?

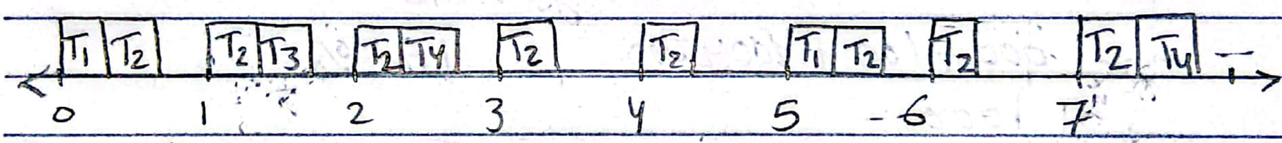
1- decrease tasks execution time
2- increase tick time
3- Task splitting "first delay" "Config"

Task	Periodicity	exec. time	first delay
T_1	5 ms	600 µS	0
T_2	1 ms	100 〃	0
T_3	10 ms	400 µS	1
T_4	5 ms	500 µS	2

→ **first delay**:

يعني عايز ال Task دي تبدأ بعد كام Tick هل

من Tick0 ولا Tick1 ولا ——



Cpu load at ⓪ → $= \frac{700}{1000} = 70\%$

Cpu Load at ②

$= \frac{100 + 500}{1000} = 60\%$

∴ Cpu load at worst case = 70%

cpu load قل من 160% إلى 70% بفعل التحكم في ال first delay

PAGE
DATE

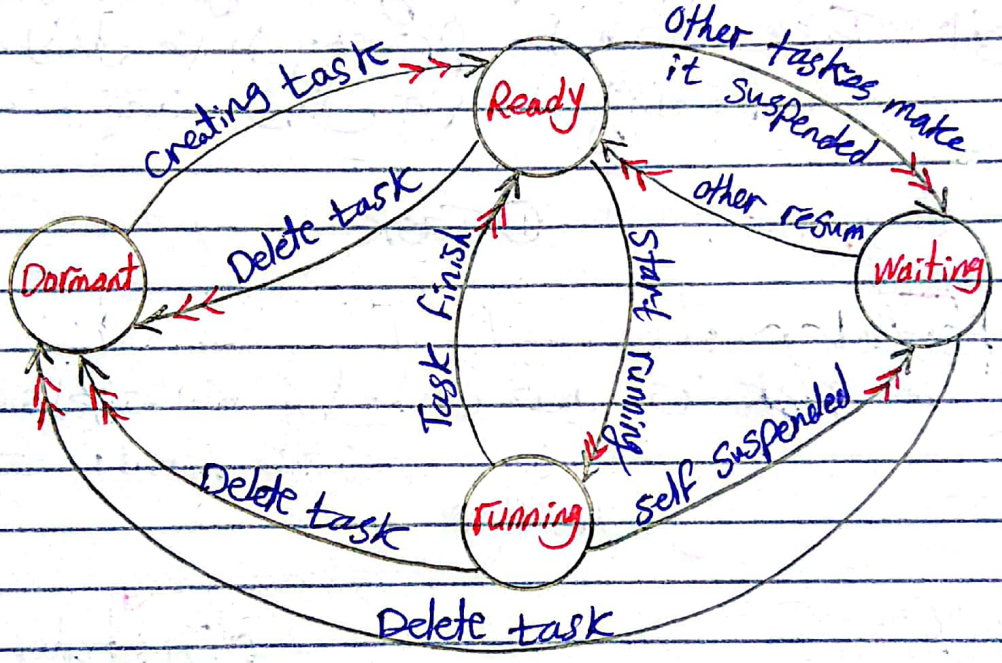
→ Task states:

1- Dormant
not created yet

2- Ready
ready to be executed

3- Suspended / Waited
interrupt / event بتستنى
حتى تخلص وترجع بعدها
ready

4- Running
بتكون أعلى Priority وكان
ال Periodicity
جه وقتها.



* task cycle →

Dormant عبارة عن Task قبل أي شيء بتكون ال
وبمجرد ما تتعمل create task بتتحول لـ Ready وتكون
مستعدة لـ execute لو ال Task هي الأعلى في
ال Priority وكان جه الدور عليها من غير ما
Running لـ Ready بتنقل من ال periodicity
ولو مثلاً جه interrupt بيخلي ال task توقف
Wating state لـ Running state من

وبعد ما الـ ISR تخلص هيكون فيه احتمالين :

① أن الـ sys مش بيدعم أن الـ Task تقطع Task تانية فبالتالي بمجرد ما الـ ISR تخلص هتتحول الـ Task من الـ waiting للـ Ready مرة أخرى ويتم تنفيذها مرة أخرى وتصبح running حتى لو فيه Tasks تانية ready وأعلى في الـ Priorty

② أن الـ sys بيدعم أن task تقطع Task تانية فبعد ما الـ ISR تخلص الـ scheduler هيشوف هل الـ Task اللي كان running وبقى waiting لأجل الـ interrupt هو الأعلى في الـ Priorty ولا فيه Task تاني أعلى منه وينفذ الأعلى في الـ Priorty .

→ RTOS Composition : ① Task

② scheduler :

هو المسئول عن تحديد أي Task هو اللي عليه الدور ينفذ

scheduler

algorithm	dispatcher
عبارة عن الطريقة اللي بتحدد بيها which Task should be executed now	المسئول عن أنه يـ execute الـ algorithm decision ويوجه الـ processor ناحية البداية اللي هيبدأ يـ execute من عندها ، مسئول عن الـ Context switching
" المخ " ، " ياخد القرار "	" العضلات " ، " ينفذ القرار "

↳ what happens if algorithm decided that a new Task should replace the current running Task ?

Dispatcher role ; "Task Control Block"

① save the current Task state TCB_1
"PC , PSW , CPU registers ..."

② restore the new Task state TCB_2

③ run the task

↳ scheduling algorithm types :

① Priorty based "most common"
معتمد على إن كل Task ← Priorty وبناءً عليه هيحدد مين هيتنفذ الأول .

② first come first serve "FCFS"
الـ Task اللي هيجي "هشغلها" create الأول هو اللي هتتنفذ الأول

③ shortest Job first "SJF"
يعني لو فيه 2 Tasks الاتنين ready وفي واحدة هتاخد وقت أقل من التانية فهيتنفذ الأقل الأول

$T_1 = 100$ µsec ✓ | $T_2 = 300$ µsec

④ shortest remaining time "SRT"
لو كان فيه 2 Taskes شغالين وحصلهم قطع لأي سبب وفيه مثلاً T_1 باقيلها 50 µsec وخلاص و T_2 باقيلها 200 µsec وخلاص يبقى هيروح ينفذ T_1 الأول لأن هي اللي باقيلها وقت أقل

⑤ Round Robin scheduling 'RR'

ودي الطريقة المستخدمة في free RTOS وهي Priorty based ولكن في حالة وجود أكتر من Task ليهم نفس ال Priorty هينفذهم مع بعض بطريقة ال Time slicing يعني مثلاً ينفذ جزء من T_1 وبعدين جزء من T_2 وهكذا فكأنهم بيتنفذوا في نفس الوقت.

→ Kernel:

it's a part of a multitasking sys responsible for the management of tasks and communication between tasks.

Kernel Composed of

objects

1- Tasks
2- Schedulers
3- Mutex
4- Message queue
5- S.W Timer

services "Functions"

1- create Task
2- delete "
3- Send massage
4- receive "

Kernel types

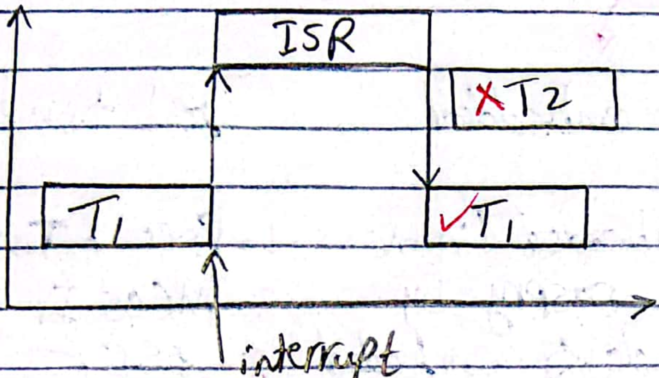
non-preemptive "Cooperate"

لا يسمح إن Task تقطع Task تانية.

Preemptive

يسمح إن Task الأعلى Priorty تقطع الأقل من ال Priorty

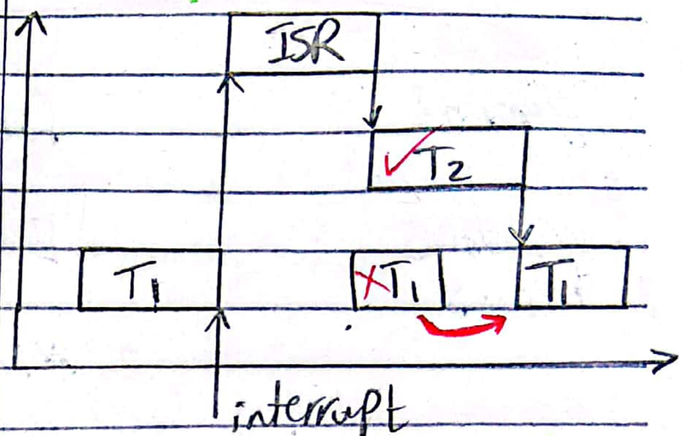
Non-Preemptive



أثناء ما T_1 كانت running
جالها interrupt قطعها وحولها
لـ waiting والـ Processor
راح ينفذ الـ ISR وبعد ماخلص
الـ T_1 اتحولت لـ Ready
مرة أخرى وعلى الرغم من
أن T_2 أعلى Priorty من T_1

ولكن T_1 هي اللي هتكمل executing
بعد الـ ISR لأن الـ Kernel
هنا مش بيسمح بعملية
القطع بين الـ Taskes

Preemptive



نفس الفكرة بعد ما الـ ISR
يخلص هيلاقي T_1 و T_2 و
T_2 أعلى من T_1 في الـ Priorty
والـ Kernel هنا بيسمح بالقطع
عادي فبالتالي هيروح ينفذ
T_2 الأول وبعدين يرجع
يكمل T_1 من مكان ما وقف
قبل الـ ISR.

Priorty

static
fixed
ومش بيقدر يغيرها
أثناء الـ O.S
running

dynamic
changable
يقدر يغيرها عادي
في الـ run time

PAGE
DATE

types of O.S

general	for embedded	real time
Ex:s		
1- Windows	1- Embedded linux	1- free RTOS
2- Linux		2- MCOS II
	2- Windows embeded	3- OSEK "Autosar"

1→ free RTOS:

it's an open source, real time operating sys kernal designed for embedded systems and microcontrollers.
it's provides a small and efficient operating sys that allows developers to create real time applications by managing tasks, scheduling and various sys resources.

⇒ Features of using free RTOS:

1- open source "بالتالى يقدر أعدل عليه براحتى"
2- free "بيكون الدعم مجانى لو محتاج دعم"

3- Preemptive or non preemptive feature
4- notification mechanism
مش بس بعتمد على الـ peripheral interrupt
notification على ان الـ event ده حدث أو لا

5- event Group
6- Priority based with Probability to be Round Robin based.

7- Inter task event Synchronisation
ان يكون فيه تزامن بين الـ Tasks وتقدر تكلم بعض
8- Software timers

9- Semaphore & mutex feature "مهم"
10- Task Communication: "التواصل بين Tasks" =

⇒ The main algorithm of free RTOS is Priority base and if there is two or more Tasks with the same Priority, free RTOS also support Round Robin scheduling "time slicing"

↳ ideal task:
بيكون Task فاضية أو فيها بعض الـ instructions بيتم تنفيذها في الوقت الفاضي اللي مفيهوش Tasks بتنفذ

↳ shared resource:
عبارة عن حاجة SW / HW بتكون shared بين أكتر من Task وبالتالي إمكانية أن يتعامل مع ACCESS من الـ Tasks دي سواء في نفس الوقت أو كل Task تعامل في access في أوقات مختلفة.

types

SW
Ex: 1- Global Variable
2- HW register "I/o, Cpu reg."

HW
Ex: 1- LCD
2- Printer

Race Condition:

Problem happens when accessing the shared resource which is non-atomic accessing.

يعني لما يكون فيه أكثر من Task ليهم نفس الـ Priority ومشتركين في shared resource زي LCD مثلاً وبالتالي

هيحصل سباق "race" بيقوم على مين هياخد الـ access واللي هيحصل أن كل واحد فيهم هياخد الـ access شوية وبالتالي الـ output هيكون مشترك بين الـ 2 Tasks.

Atomatic access Vs Non-Atomatic access

Atomatic access	Non-Atomatic access
عبارة عن action بيحصل من غير ما يعمله interrupt أو أي task تانية تقطعه حتى لو أعلى Priority لأنه مش بيستغرق غير 1 cycle فبالتالي في كل الظروف لازم الـ Processor يخلص الـ cycle الأول قبل ما يعمل check على وجود interrupt من عدمه.	عبارة عن الـ action اللي ممكن يتقطع عادي وهو بيـ execute بسبب interrupt أو حتى بسبب another task وده بياخد more than one clock cycle فبالتالي إمكانية قطعه بعد أي cycle من الـ cycles بتاعته.
Ex: Inc: PortA = 1; In assembly: out PortA, #1	Ex: setBit(PortA, 0x01); هيتفك لكذا سطر assembly وكل سطر يعتبر 1 cycle

PAGE
DATE

طب إزاي نحل مشكلة الـ Race Condition اللي بتحصل للـ shared resoure ؟

① **make Your access Atomic** "not always Valid"
يعني الـ access مياخدش أكثر من 1 cycle ولكن ده صعب ومش متاح دائماً.

② **Disable Global interrupt**
ولكن يكون لوقت قصير جداً" ولكن من 2:4 clk cycles
وده بيسمى بالـ critical section

EX:

Disable Global interrupt —(Free RTos Func)→ EnterCritical Section( );

| critical section | //code

Enable Global interrupt
↳ Func name at free RTos → Exit Critical Section( );

فبالتالي الـ critical section ده مفيش أي interrupt يقدر يقطعه.

③ **Create a Busy flag**

يكون عبارة عن Global flag ولكن ليه initial value = 1
فلما يدخل الـ Task1 ياخد الـ flag ده يخليه
صفر = ⓪ بحيث أن الـ Tasks التانية متعرفش
تاخده وبعدما ينفذ الـ Task1 يرجع يسيب الـ flag
بأن يخليه تاني = 1 وبالتالي يروح ينفذ باقي
الـ Tasks . والـ flag ده بيسمى
بالـ Semaphore

Ex: Global_flag = 1;

```
main(){
   ≡
}
Void Task1 (Void){
  while(1){
    if(Global_flag == 1){
      Global_flag = 0;
        ≡ //Code.
      Global_flag = 1;
    }
```

→ Task Take the flag of the shared resource

↳ release the flag after finish the Task.

↳ **What is Semaphore?**

it's an integer variable that is initialization with the number of resources that present in the Sys and is used for process synchronization.

it's types

binary

value → 0 or 1

used for shared resource protection "solution"

Counting

value → 0, 1, 2, 3 ...

used for resource Limit

PAGE
DATE

↲ لذن إستخدام ال Counting أكثر من ال binary فمن الغالب هستخدم ال Counting ولو تحتاج تخليه binary هتعمله
1 resource ←
initial value = 1 ←

↳ operation on semaphore :

1- Decrease / Pend / Take
يعني تاخد ال semaphore لل Task دي بحيث مفيش Tasks تانية تعرف تقطعها وبيكون في البداية قبل عملية ال accessing لل shared resource .

2- Increase / post / Give
يعني خلاص ال Task خلصت وسابت ال semaphore وأي Task تقدر تدخل تاخد ال semaphore وتستخدم ال shared resource .

↲ وفي الحالتين 0 ← semaphore unavailable
أي قيمة > 0 ← semaphore available

↲ نلاحظ أن ال semaphore هي block of memory اسمه ECB ← "Event control Block" عشان يخزن فيه مجموعة من ال data زي :
1- Type "binary or Counting"
2- current value
3- Waiting Tasks to take the semaphore.

PAGE
DATE

How to use semaphore of free RTOS ?

① include semaphore header file:

```
#include "semaphore.h"
```

② Define / create semaphore name:

```
xSemaphoreHandle name;
```

Ex:

```
xSemaphoreHandle LCD_sem;
```

③ Create a binary semaphore;

Ex:

```
(V)SemaphoreCreateBinary(LCD_sem);
```

(V) return void — LCD_sem ← semaphore name

For create a Counting semaphore:

```
(X)SemaphoreCreateCounting(MaxCount, initialCount);
```

(X) return data base ... mc

↳ return: 1- NULL → لم يتم إنشاء الـ semaphore

2- another value → handle to the created semaphore

Ex:

```
LCD_sem = xSemaphoreCreateCounting(1,1);
if(LCD_sem != NULL){
    // semaphore created successfully
}else{
    // semaphore not created
}
```

PAGE
DATE

④ Take the Semaphore:

xSemaphoreTake(semaphore , Ticks to wait)

↳ return ; 1- PdPASS → لما يتعمله Take بشكل صحيح

2- PdFAIL → لما ميعرفش يعمله Take

Ex:

```
u8 semstate = xSemaphoreTake( LCD_sem,0);
if ( semstate == PdPASS){
    // LCD functionality
}
else {
```

معرفش ياخد ال semaphore وهستنى لحد ما ال Ticks to wait تخلص ولو معرفش ياخد ال semaphore وهو wait بردو هيدخل ال else ينفذ اللى فيه .

```
}
```

⑤ Give the Semaphore

xSemaphoreGive(semaphore)

↳ return → PdPASS سابه بنجاح
↳ PdFAIL فشل بسبب ال semaphore

Ex:

```
u8 semGive = xSemaphoreGive(LCD-sem);
```

وممكن بعد ما اعمل check على ال semGive حيث لو كان = PdPASS يعني معنى Task كامله واخدها .

PAGE
DATE

↳ Semaphore Problems :

1 Starvation :

"يعني المجاعة" ← لو عندي مجموعة من الـ Tasks وليهم Priorities مختلفة وكلهم عاملين access على shared resource فالـ low Priority بنسبة كبيرة مش هتعرف تعمل access على الـ resource

الحل : الحل معتمد علينا كـ designers فانا نتأكد ان الـ Priorities والـ periodicities متوزعين بشكل يضمن ان كل الـ Tasks تـ run بشكل صحيح

2 Dead lock :

مشكلة بتحصل بسبب ان فيه 2 Tasks مثلا كل واحد waiting للتاني يخلص ومستني بسبب الـ semaphore

EX :

Task1	Task2
{	{
Take Semaphore A;	Take semaphore B;
Take semaphore B;	Take Semaphore A;
⋮	⋮
Give Semaphore B;	Give semaphore A;
Give semaphore A;	Give Semaphore B;
}	}

← في الـ EX : Task1 & Task2 نفس الـ Priority

PAGE

DATE

← فبالتالي ال Kernal هيتعامل معاهم بناءً على Round Robin فهينفذ هنا شوية وهنا شوية.

← Task1 هتاخد sem_A وبعدها Task2 هتاخد sem_B وبعدين Task1 هتطلب sem_B وكده هتعمل deadlock مع Task2 فهتروح على ال waiting لحد ما Task2 تسيب sem_B ونفس الكلام Task2 هتطلب sem_A فهو مع Task1 فكدا بردو Task2 هتروح ال waiting وهيفضلوا كدا في ال waiting ويوصلوا للحيطة.

الحل: 1- ال Task تاخد كل ال Semaphore اللي عايزها من أول الكود بتاعها

2- لو الحل ① منفعش يبقى هنلجأ للترتيب وهو أن كل ال Tasks اللي طالبين نفس ال resource يكون ال Semaphores مطلوبة بشكل مترتب عندهم

EX:

Task1
{
Take sem A
Take sem B
=
}

Task2
{
Take sem A
Take semB
=
}

فبالتالي لو Task1 هي اللي خدت ال sem A فلما ييجي ينفذ Task2 هيلاقي أنه بيطلب sem A اللي هو مع Task1 أصلاً فال Kernal هينقل Task2 لل waiting لحد ما Task1 يسيب ال sem A

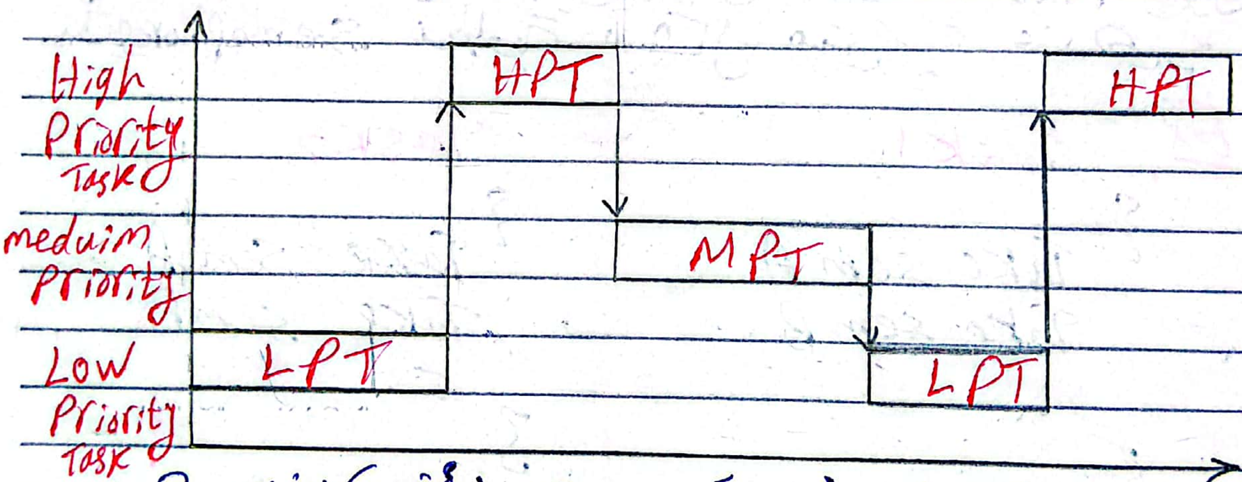
PAGE
DATE

٣ والـ Task يتسيب الـ semaphores تسيبها بعكس الترتيب اللى أخدتها بيه .

Ex:

```
Task 1
{
  Take  Sem A
  Take  Sem B
     ≡
  Give  Sem B
  Give  Sem A
}
```

3 Priority inversion :



← هنا الـ LPT بدأت تـ run لأنها كانت هى بس اللى Ready ولما بدأت خدت الـ semaphore A

وبعدين جات HPT وبدأت ينفذها والـ LPT دخلت الـ waiting والـ Processor بينفذ الـ HPT لحد أنها بتطلب semaphore A اللى هو مع LPT فبالتالى

PAGE
DATE

الـ HPT هتنتقل للـ waiting وتستنى الـ LPT لحد ما تسيب الـ semaphoreA فالـ secheduler هيجي فيلاقي

الـ LPT انتقلت من الـ waiting للـ Ready وكذلك الـ MPT Ready فهينفذ الـ MPT الأول ولما تخلص هيرجع ينفذ LPT لحد ما تسيب الـ semaphoreA والـ HPT تخرج من الـ waiting وتاخد الـ Sem-A

← يبقى المشكلة هنا أن MPT أتعملها execute قبل الـ HPT وهي دي الـ priority inversion

الحل: هنا الـ RTOS هيتدخل ويدينا كذا حل:

① **Priority Caling:**

هنا بنعمل جدول كدا وكل الـ Taskes اللي عايزة تاخد الـ semaphore المعين ده تكتبها في الجدول والـ semaphore ياخد أكبر priority فلو هنطبق على المثال ده:

Task	Priority
1	3
2	1
3	7
4	10

semaphore priority = 10

فكدا أي Task هتحتاج تاخد الـ semaphore الـ priority بتاعتها هتتغير لـ 10 يعني High priority

فنكدا أي Task أقل منها في الـ priority مش هتقطعها

وبمجرد ما الـ Task تخلص / تسيب الـ semaphore هترجع لـ priority بتاعتها القديمة مرة تانية.

PAGE
DATE

← عيوب الحل ده: أنه بيغير الـ Priority حتى لو مش محتاج أغيرها يعني لو مفيش MPT فهيغير الـ Priority برده وده بيسبب تعب للـ processor.

② Priority inheritance:

← في الحل / الطريقة دي حل العيب اللي ظهر في الطريقة الأولى هنا مش هيغير الـ Priority إلا في حالة

أن الـ "HPT" هو اللي طلب الـ semaphore وأن الـ semaphore ده موجود مع "LPT" فبالتالي

الـ "LPT" هتتعلى وتبقى "HPT" لحد ما تـ Give الـ semaphore وترجع low مرة تانية.

← في الحالة دي مش بيكون اسمه semaphore ولكن بيصبح اسمه "Mutex"

↳ mutex ⟶ semaphore + Priority inheritance

↳ inter Task event synchronisation:

← لما بيكون عندي 2 Tasks وفيه action / event معين Task1 بتعمله وبناءً على الـ event ده حصل أو محصلش هياخد Task2 قرار

← مثلاً لو عندنا Task خاصة بالـ KeyPad وبناءً على ضغطة زرار معين فيه Task تانية هتعمل Toggle للـ Led

← هل الsynchronisation هو نفس الCommunication ؟

لأ، لأنه فيه إختلاف بينهم حيث أن الsynchronisation بيهتم بتفاصيل True or False يعني الevent حصل أو محصلش

بينما الCommunication بيكون ليها أكتر تفاصيل أكتر

EX:

```
Void ADC_Task (Void){
  //perform Initialization
  while (1) {
      //start conversion
      // enable ADC_interrupt
    u8 state = xSemaphoreTake (ADC);
    if (state == pdpass ) {
          // Code
    }
      vTaskDelay (1000);
}
```

```
Void ADC_ISR(Void)
{
   u8 state =
   xSemaphoreGive
   (ADC);
}
```

← داخل الISR إستخدمنا الsemaphore كـ notification إن الConversion خلص وبالتالي مش هينفذ الfunc بتاعت الADC غير لما الConversion خلص من غير ما يعمل الpoling to sys لأن الADC_Task هتفضل فى الwaiting لحد ما الConversion خلص ويروح للISR فيبعت الsemaphore ويرجع للADC_Task يكمل باقي الfunc عادي

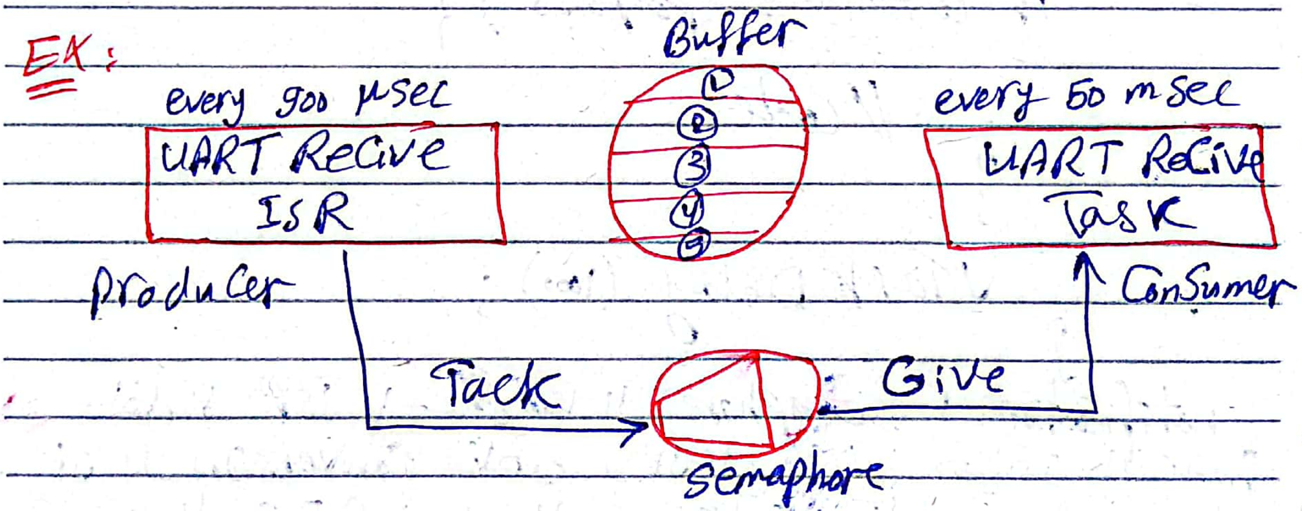
← هنلاحظ في المثال السابق أن الـ semaphore في
البداية مش متاح أصلاً على عكس الـ shared resource
يكون initially متاح .

← طريقة الـ synchronisation دي بتسمى wait signal
يعني Task مستنية signal من Task تانية
وبنستخدم معاها الـ Binary semaphore ولو عايزين
نستخدم الـ Counting semaphore يبقى نعملها initially
بـ Zero

← Producer Consumer

← دي طريقة تانية للـ Synchronisation بنستخدم معاها
الـ Counting semaphore وبنستخدم كمان Buffer عشان
نخزن فيه الداتا

EX:



← الـ ISR بتستلم data كل 900 µsec وبتعمل حاجتين لما تستلم
الـ data ← بتعمل increament للـ semaphore
← بتعمل save للـ data داخل الـ Buffer
← الـ UART Task بتيجي كل 50 msec تحاول تاخد
الـ semaphore لو نجحت يبقى فيه داتا هتقرأها ولو لا
يبقى هتستنى لحد ما تاخد الـ sem وتقرأ الداتا من
الـ Buffer

← ملخص لإستخدامات الـ Semaphore :

1 Binary semaphore :

↳ shared resource protection (initially 1)
بيكون متاح من البداية

↳ event synchronisation (initially 0)
بيكون مش متاح من البداية زي الـ wait signal

2 Counting semaphore :

↳ resource Limit (initially = max Count)

↳ event synchronisation (initially = zero)
بيبدأ من 0 ← max count زي الـ producer consumer

↳ InterTask Communication :

← هنا بيكون Task عايزة تكلم أو تبعت massage لـ Task تانية فهنستخدم الـ Kernel services وهنستخدم الـ Queue

← بنقدر من خلالها نخلي ← Task تكلم Task
// // ISR
ISR // Task

← الـ Queue عبارة عن memory بـ organized فيها الـ data بطريقة الـ FIFO يعني [First IN First out]

← عشان نستخدم الـ Queue لازم نعرف حاجتين :
1- item size ← بـ حجم الـ massage
2- Length ← يعني الـ Buffer يقدر يشيل كام massage

PAGE
DATE

notes :

١- الـ Queue من النوع Many to 1 Communication
يعني بعدما الـ TASK تقرأ الـ massage من الـ buffer
بيعملوا clear تلقائي فبالتالي مفيش غير massage
واحدة اللي تستطيع انها تقرأ الـ massage

٢- نوع الـ massage اللي بنباصيها للـ Void Pointer
لأن الـ FreeRTOS مش عارف أنا هباصي massage
حجمها أي فهو هياخدها ويعملها Casting حسب الـ Size
اللي أنا هحطه .

٣- الـ Queue بيعمل Pass by Value وليس by reference
لأنه مش عارف حجم الـ massage ولا الـ address
وإستخدام الـ FreeRTOS للـ Pass by Value مميزتين :
1- save local variable
2- save data before Pointer manipulation

API of Queue:

1 Define the Queue:
↳ XQueueHandler name ;
Ex: XQueueHandler Buffer;

2 Create the Queue:
↳ XQueueCreate(length, Itemsize);
return: → Null: Queue is not created as No memory
→ Another Value: handle to the created Queue

Ex: u8 state = XQueueCreate(6, sizeof(u8));

PAGE
DATE

3 sending to the Queue:
↳ XQueueSend (XQueue, *pvItem, xTickwait);
return: 1- PdPASS: massage is sent
2- PdFAIL: Queue is full

4 Receive from Queue:
↳ XQueueReceive (XQueue, *pvItem, XTickwait);
return:
↳ PdPASS: massage is read.
↳ PdFAIL: Queue is empty

EX: u8 state = XQueueReceive (Buffer, &var, 0);

↳ Mail Box:

1 to many communication بس هتلاقيه نفس فكرة الـ Queue
مبالتالى 1 يقدر يحط رسالة في الـ mail Box و أكتر
من Task تقدر تقرأها .

charactristics:
1- has capacity for only one massage
2- after the Tasks Read the massage
it's not cleared
it's only cleared by the sender Task.

3- Can be used in broad casting
تقدر تكلم كل الـ sys

→ API of Mail Box:

1. Define the mail box
   هو هو نفس الـ Queue

2. Create the mailBox:
   Same as the Queue **with length = 1**

Ex:
`u8 mailbox = xQueueCreate(1, sizeof(u8));`

3. Sending to the mail box:
   → `xQueueOverwrite(xQueue, *pvItem);`

Ex: `u8 state = xQueueOverwrite(mailBox, &var);`

4. Receiving from the mail box:
   هنا مش بيحصل clear للـ massage بعد ما بيتم قراءتها
   → `xQueuePeek(xQueue, *pvItem, xTicksToWait);`

→ إزاي يا تستخدم الـ Queue نعمل many to many communication ؟

نعمل آلاف من Queue ونبعت لكل واحد نفس الـ massage بحيث لو الـ Task قراها ويمسحها clear تكون موجودة في الـ Queue التاني وينفع Task تاني تقراها وهكذا.