Systèmes Temps-Réel

2A Électronique Communicante et Systèmes Embarqués

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Objectifs de l'enseignement

Connaissances

- Comprendre les services fondamentaux proposés par un système d'exploitation
- Comprendre les contraintes liées au temps-réel
- Comprendre le rôle d'un ordonnanceur
- Comprendre les contraintes liées au développement d'un RTOS
- Comprendre et utiliser les services proposés par un RTOS

Savoir-faire

- Maîtriser les services et outils logiciels proposés par une solution du marché
- Représenter et comprendre des besoins avec des schémas représentant les outils du RTOS
- Concevoir une application logicielle embarquée portée sur RTOS

Plan de la matière

Cours (5h CM, D. Boudier)

- Introduction
- Mécanismes des Systèmes d'Exploitation Temps-Réel
- Programmation des RTOS : Application à FreeRTOS
- Tour d'horizon des algorithmes d'ordonnancement
- Choisir son Système d'Exploitation Temps-Réel
- Contenu bonus

TP (15h TP, D. Boudier)

- FreeRTOS en mode coopératif
- FreeRTOS en mode préemptif
- Outils de FreeRTOS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Examen écrit 1h30

Questions générales pouvant porter sur n'importe quel aspect vu en cours/TP

Analyse d'une solution technique (processeur et RTOS inconnu) → Voir exemples fournis

Concevoir un modèle logiciel générique à partir d'un cahier des charges

Examen pratique 1h30

Implémenter une solution logicielle à partir d'un cahier des charges

Utilisation de FreeRTOS sur Microchip PIC32

Le système d'exploitation temps-réel utilisé en TP (et en guise d'exemple sur les diapos de cours) est FreeRTOS.

Il s'agit d'un RTOS très populaire, facilement modulable aux besoins, adaptés aux petits systèmes (le binaire du noyau fait entre 4 KB et 9 KB) et portable sur près de 40 architectures différentes.

Distribué sous licence MIT, il est donc libre et open-source.

Technologies utilisées

Nous porterons le RTOS sur un processeur du concepteur ARM issu de la famille Cortex-M.

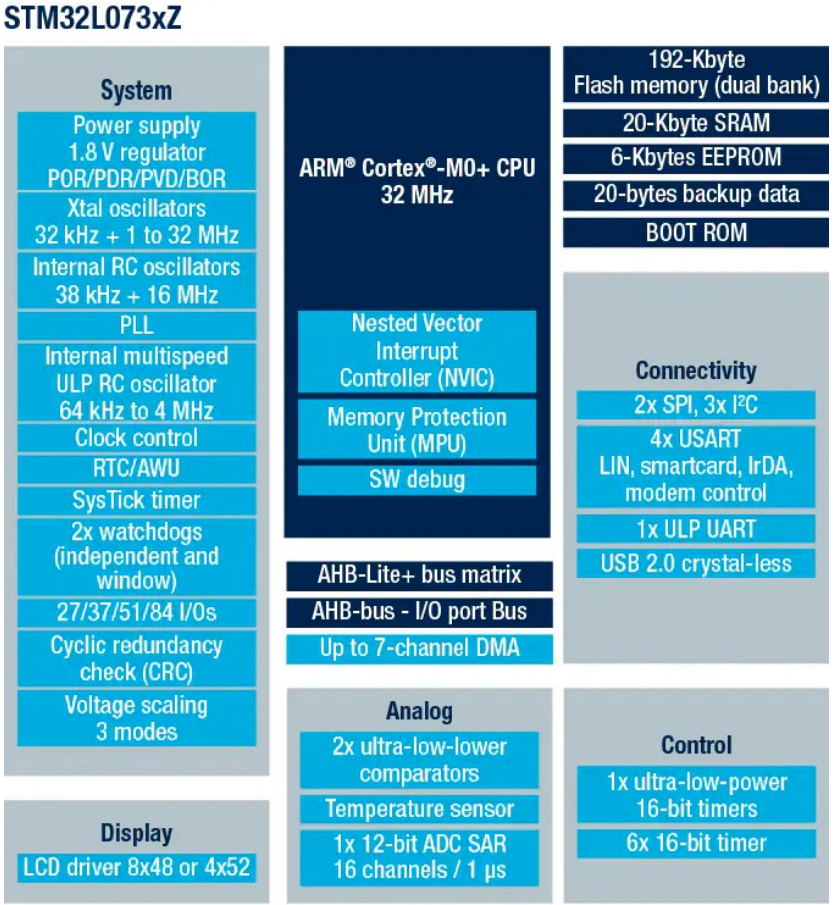
Plus précisément, nous utiliserons un MCU STM32L073RZ conçu par STMicroelectronics, et se basant sur un cœur ARM Cortex-M0+.

Cette solution est disponible sur une carte d'évaluation de la gamme Nucleo-64.

Carte Nucleo : Nucleo-L073RZ

MCU : STM32L073RZT6U

IDE : STM32CubeIDE



Toutes les ressources se trouvent sur Moodle, en accès libre :

https://foad.ensicaen.fr/course/view.php?id=840

Vous y trouverez :

- Le support de cours en PDF
- L'archive de TP (sujet, fichiers sources)
- Les outils à installer pour travailler sur votre ordinateur

Sources

- Introduction aux systèmes temps réel, C. Bonnet, I. Demeure, HERMES Sciences Publications, 1999. ISBN 2-7462-0016-3

- Systèmes d'exploitation 3ème ed., A. Tanenbaum, Edition Pearson Education. ISBN 978-2-7440-7299-4

- OSDev.org, http://wiki.osdev.org/Main_Page

- The Definitive Guide to the ARM Cortex-M0, J. Yiu, Newnes editions, 2011. ISBN 978-0-12-385477-3

- Systèmes d'exploitation temps réel, G. Frey, cours EnsiCaen 3A info/instru, 2013.

- Systèmes temps-réels, R. Demoment, NXP Semiconducteurs, 14460 Colombelles, 2010.

- An analysis and description of the inner workings of the FreeRTOS kernel, R. Goyette, Carleton University, 2007.

- Selecting an embedded Operating System, Colin Walls, Technical Marketing Manager, Mentor Embedded, webinars MentorGraphics, 2013.

- 2019 Embedded Markets Study, EETimes, embedded, AspenCore, 2019.

- Site de FreeRTOS, une mine d'or : https://www.freertos.org/

Un grand merci à Basile Dufay, Maître de Conférences à l'ESIX, pour le partage de ses sources de CM RTOS !

SYSTÈMES D'EXPLOITATION
OPERATING SYSTEM (OS)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Définition
Historique
Marchés

Définition

Un **système d'exploitation** ou ***Operating System* (OS)** est un logiciel qui pilote un système informatique en contrôlant ses ressources.

Cela inclut :

- Gestion des fichiers (si existence d'un *File System*) : organisation, implantation
- Gestion de la mémoire : par exemple, gestion de l'unité de contrôle de la mémoire (MMU)
- Gestion de la sécurité et des politiques d'accès : multi-utilisateurs, multi-tâches
- Gestion des processus : création, planification, coopération, dépendance, …

En revanche, un OS n'est pas :

- Un logiciel applicatif (traitement de texte, compilateur, navigateur Web, …)
- Une interface graphique (GUI, CLI)

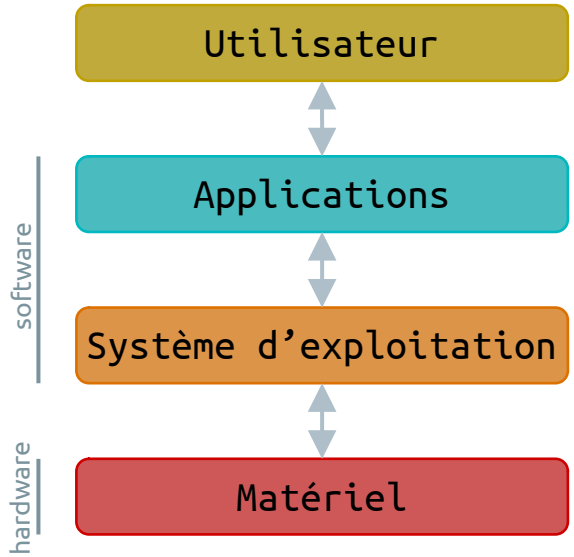
On peut considérer l'OS comme étant un intermédiaire entre les applications logicielles et les ressources matérielles de la machine sur laquelle il fonctionne.

Un OS peut être vu comme un gestionnaire de ressources :

Gestion des ressources matérielles (CPU, mémoire de masse ou de travail, périphériques, …) pour les attribuer de manière optimale entre les différents processus qui les demandent.

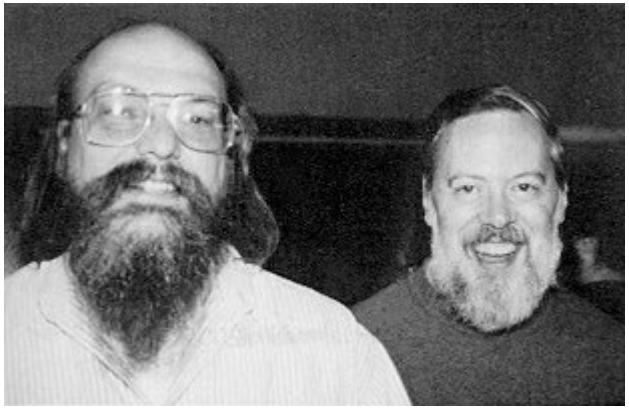
Il peut aussi être vu comme une couche d'abstraction :

Il masque les mécanismes de fonctionnement de la machine dans le but de présenter une interface simple au développeur.

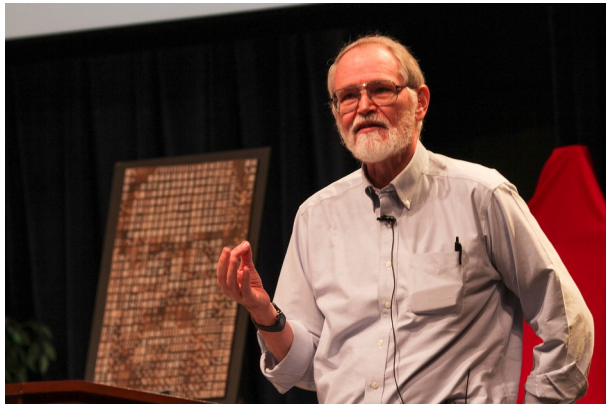


Historique

- Années 40 et 50 : pas d'OS, les systèmes sont mono-tâche et mono-utilisateur
- Années 60 : le MIT crée le premier OS, Multics. Il est multi-programmes, multi-utilisateurs
- Années 70 : UNIX développé par Thompson et Ritchie (Bell Labs), OS pour lequel le C est créé (Kernighan et Ritchie)



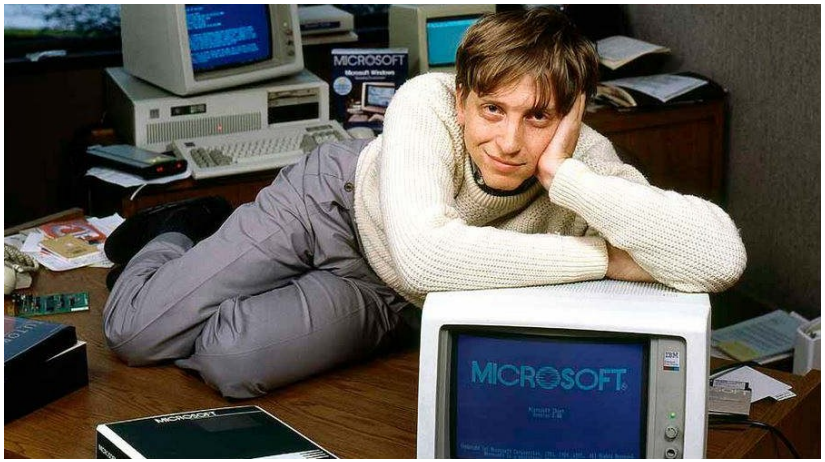
Ken Thompson et Dennis Ritchie.



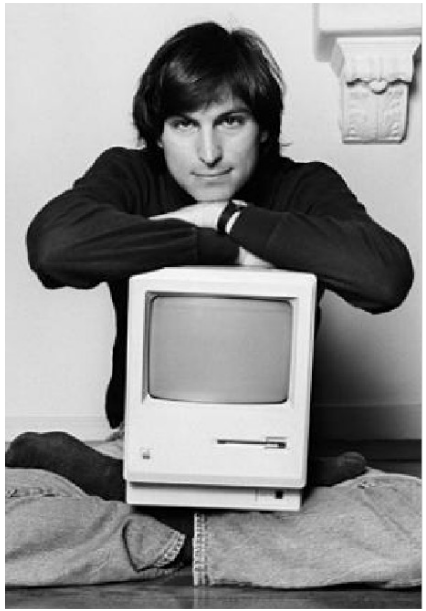
Brian Kernighan,
Hommage à D. Ritchie en 2012 (Bell Labs).

Historique

- Années 80 : IBM et Microsoft fondent le MS-DOS
- En parallèle : Xerox et Steve Jobs développent Xerox Star, abandonné puis réadapté pour Macintosh



Bill Gates

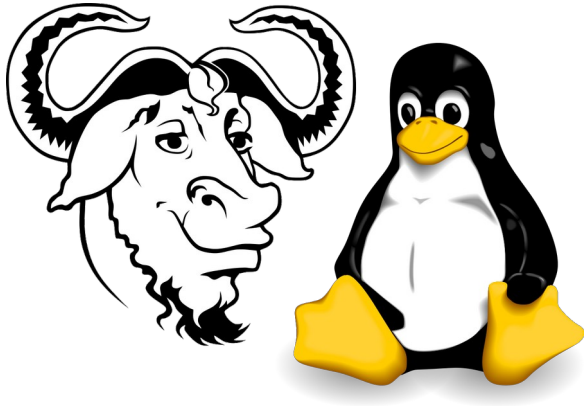


Steve Jobs

Historique

- 1983 : Stallman (MIT) crée GNU. 1er sous licence libre (la GNU GPL) mais c'est un OS sans noyau
- 1991 : Torvalds développe le noyau Linux (licence GPL)
- 1994 : GNU et Linux s'associent pour donner naissance au premier OS 100 % libre : GNU/Linux

Richard Matthew Stallman



Logos GNU et Linux

Linus Torvalds

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Évolutions technologiques

Années 1960

Première génération : Système de traitement par lots

Exécution de grands calculs successifs, peu d'intervention utilisateur

Années 1970

Deuxième génération : Systèmes multi-programmés

Exécution des programmes par *scheduling* (ordonnancement ou planification)

Troisième génération : Systèmes en temps partagé

Années 1980

Le *scheduling* cherche à répondre aux demandes de plusieurs utilisateurs en communication directe

Années 2000

Quatrième génération : Systèmes temps-réel

L'objectif est de garantir l'exécution des tâches en un temps donné

Années 2010

Cinquième génération : Système distribué

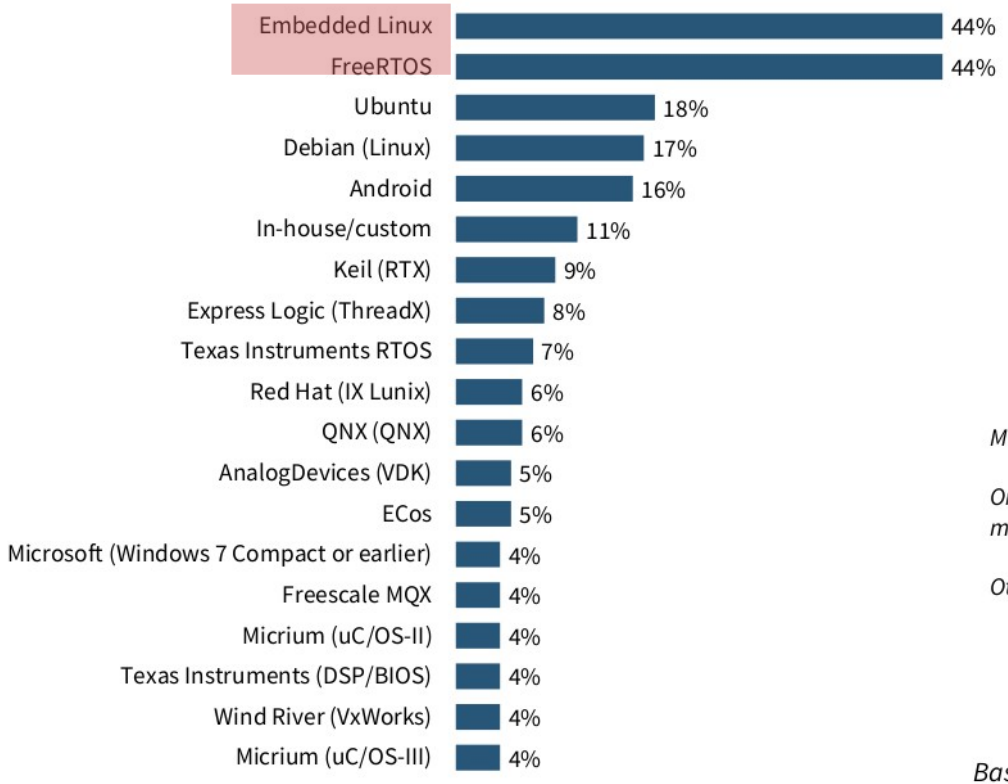
Gestion des ressources de plusieurs ordinateurs en simultané, via réseau informatique

Ce groupe de machine est vu comme une unique machine virtuelle, aux capacités importantes

SYSTÈMES D'EXPLOITATION / OPERATING SYSTEM (OS)

Étude du marché 2023

Presented by : EETimes, embedded



Multiple responses allowed

Only those with 4% or more total mentions shown

Other = 7%

Base = Those who will use an OS (566)

RÉALISER UN SYSTÈME MULTI-TÂCHES POUR L'EMBARQUÉ

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Pourquoi ajouter un OS et complexifier une solution quand on peut programmer des micro-contrôleurs avec un simple programme ?

→ Certains systèmes doivent être capables de gérer plusieurs tâches simultanément

Ces systèmes sont nommés « **systèmes multi-tâches** ».

Dans un tel système, les tâches exécutées peuvent être indépendantes, coopérantes (échanges d'informations) ou aussi en compétition (besoin de synchronisation).

Ces tâches peuvent être crées statiquement (au démarrage du système) ou dynamiquement (création et destruction à la volée).

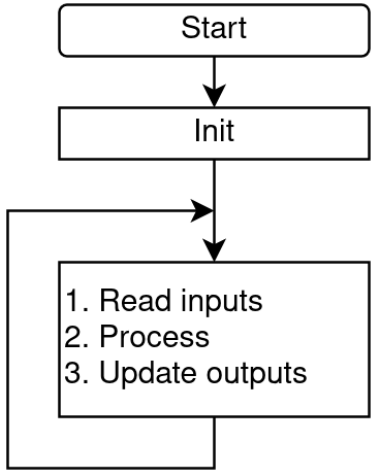
Toutes ces caractéristiques sont difficilement gérables avec un simple programme séquentiel.

De l'utilité des tâches

Cas le plus simple : traitement séquentiel

Les entrées (ex : ADC) sont toujours à jour (valides).

→ Le programme sait quand il doit interroger les E/S



On complique un peu

Les entrées ne sont pas forcément à jour.

→ Comment le programme peut-il savoir quand démarrer un traitement ?

→ Le programme doit-il respecter des échéances ?

On complique encore plus

Plusieurs évènements peuvent se produire indépendamment et n'importe quand

→ Comment partager le CPU entre les différents traitements liés à chaque évènement ?

→ Analysons donc différentes stratégies de gestion des tâches.

Scrutation cyclique (*polling*)

Avantages

- Simple à mettre en œuvre

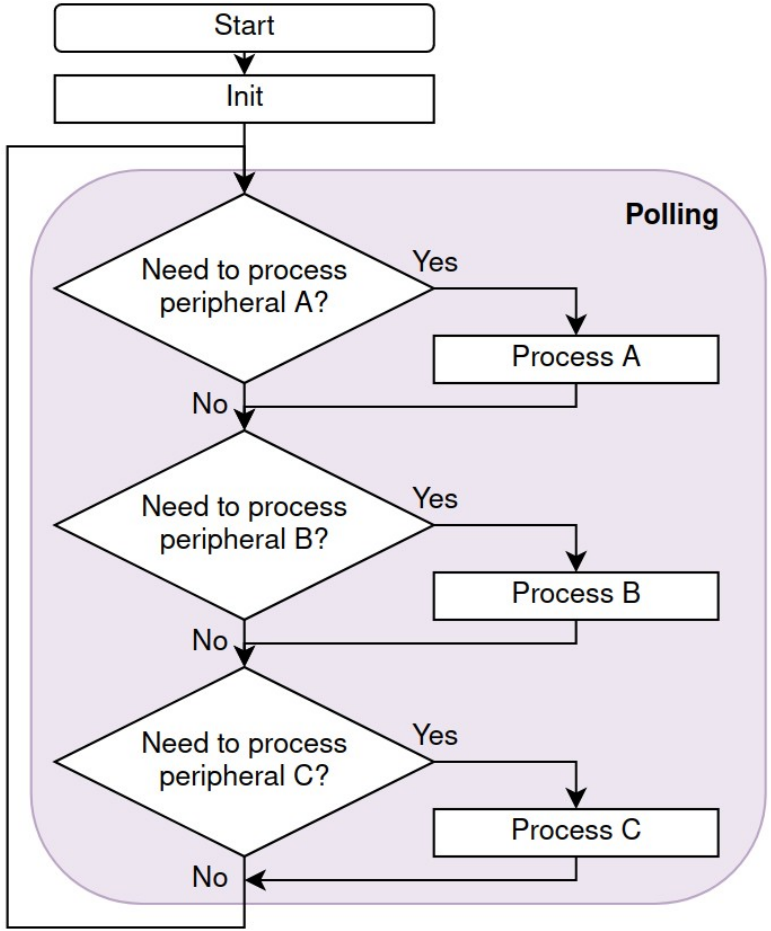
- Temps de réaction facile à déterminer (temps au pire cas)

Inconvénients

- Beaucoup de périphériques = temps de réaction qui augmente et fréquence de scrutation qui diminue

- Scrutation effectuée par le CPU même si aucun traitement requis

- Peu évolutif



Gestion par interruptions

Avantages

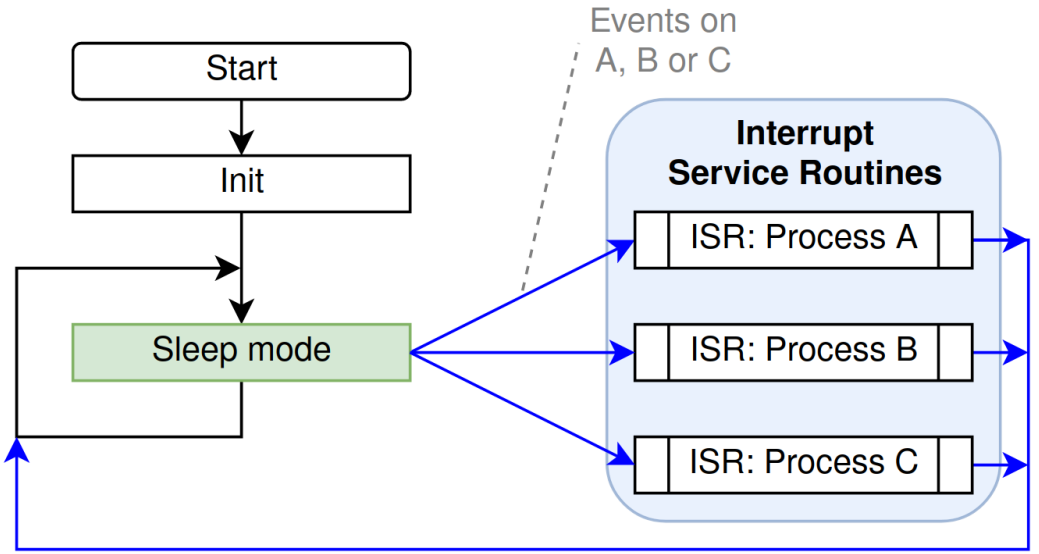
- Temps de réaction réduit par rapport au *polling*

- CPU en *Sleep mode*

Inconvénients

- Beaucoup de périphériques
= temps passé dans les ISR qui augmente
= risque de manquer un évènement

- Peu évolutif



Combinaison *polling* + interruptions

Avantages

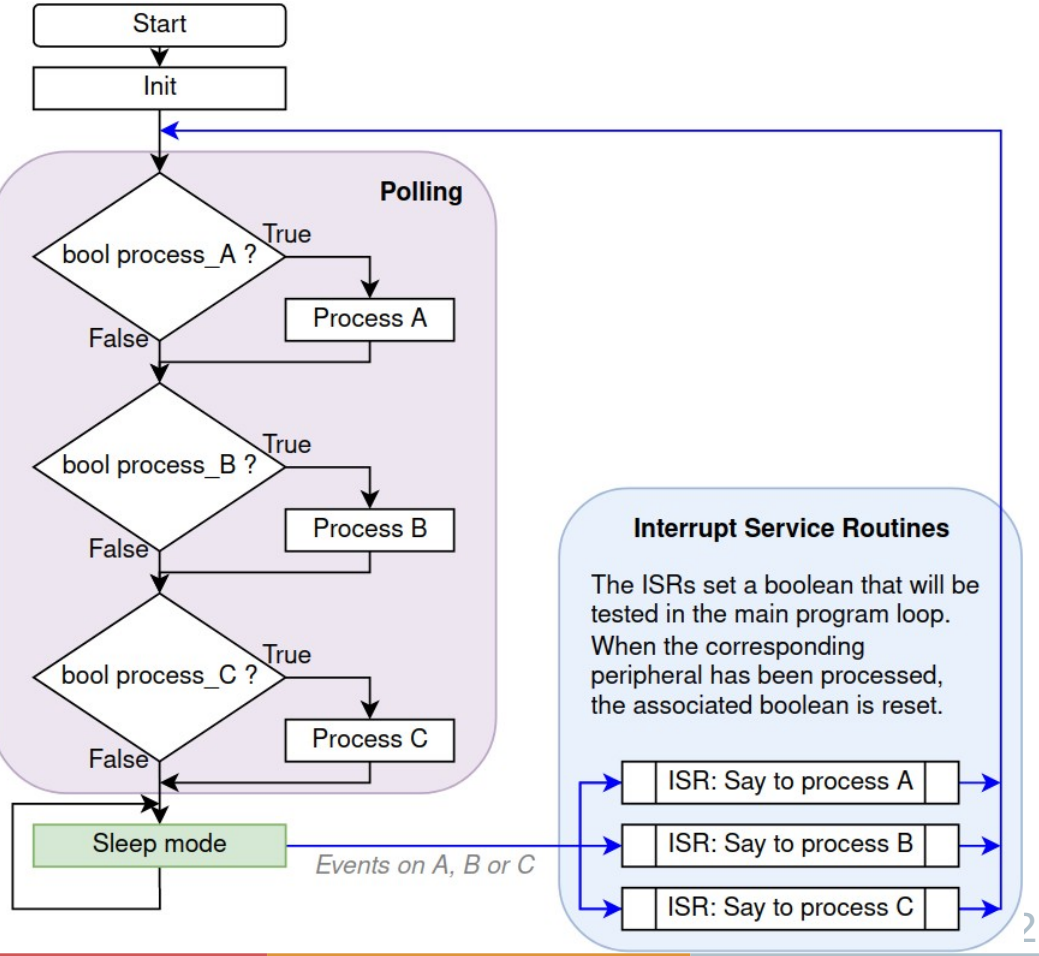
- Moins de temps dans les ISR → IRQ de faible priorité servies plus facilement

- CPU en *Sleep mode*

Inconvénients

- Beaucoup de périphériques
= temps de réaction qui augmente
= risque de traiter trop tard

- Très peu évolutif



Séquenceur (*Offline scheduler*)

Aussi appelé ***Round Robin***

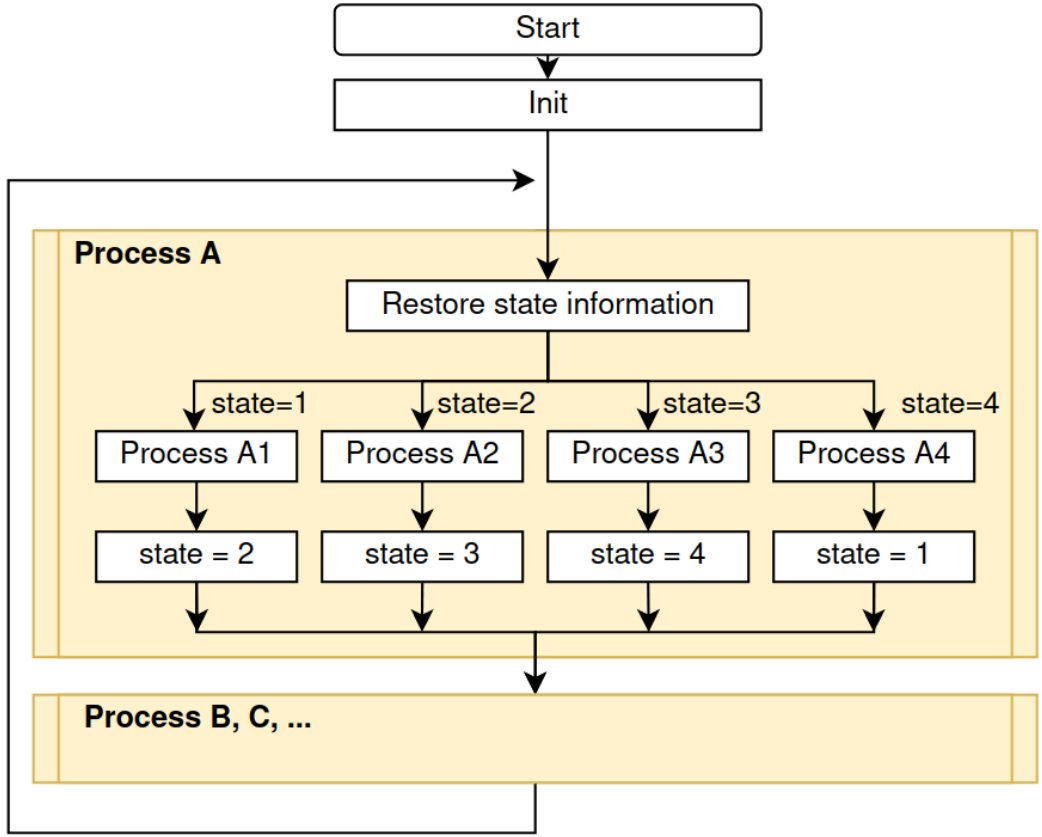
Avantages

- Meilleure réactivité

Inconvénients

- Si application trop complexe, impossible de fragmenter les processus à la main

- Très peu évolutif (tout re-partitionner pour intégrer un nouveau périphérique ?)



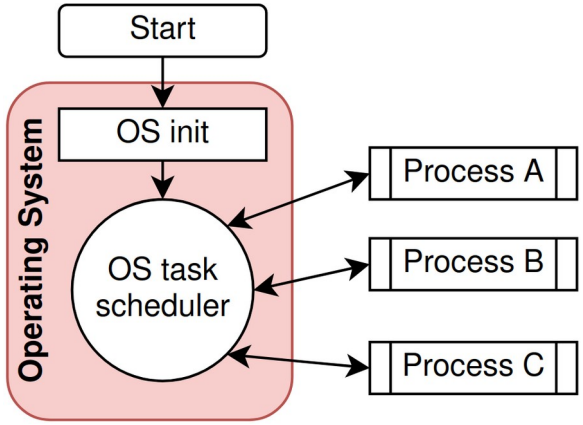
Système d'exploitation temps-réel

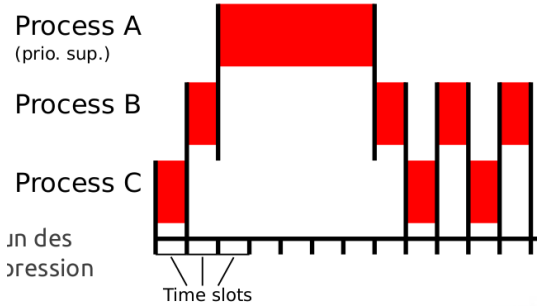
Avantages

- Programmation multi-tâches
- Simplicité d'utilisation
- Échéance garantie *a priori*

Inconvénients

- Surconsommation des ressources matérielles (CPU, mémoires données et programme)





Le temps CPU est découpé en tranches (*slots*) allouées à chacun des processus (en fonction de leur priorité éventuelle).

VOCABULAIRE

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Système temps-réel = système déterministe

Définition : **temps-réel**

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés.

« Système temps réel », Wikipedia

→ La spécificité d'un système temps-réel est donc de respecter des contraintes temporelles.

Définition : **déterminisme**

→ Le **déterminisme** est la caractéristique fondamentale des systèmes temps-réels.

Déterminisme logique : les mêmes données en entrée d'un même système produisent les mêmes résultats.

Déterminisme temporel : le systèmes doit respecter des contraintes temporelles (échéances).

Pour qualifier un système de « temps-réel », il faut prouver qu'il respecte toutes les contraintes.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Définition : concurrence

Un système peut avoir à gérer des évènements et/ou des activités simultanément. Ces évènements et activités sont dites « en concurrence ».

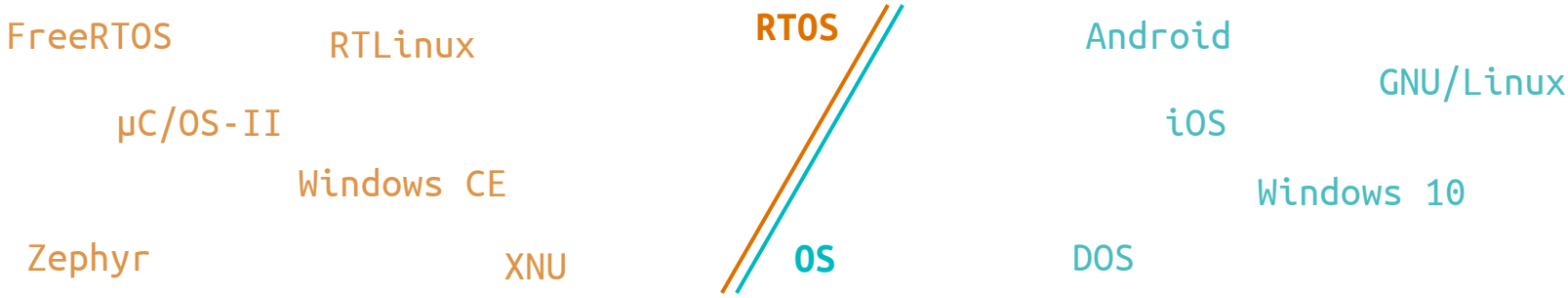
Attention : on parle parfois de traitement parallèle, mais pour les RTOS il ne s'agit très généralement pas d'une réalité physique.

Définition : préemption

Il s'agit de la capacité du système d'interrompre une tâche en cours au profit d'une autre.

Ce concept est déjà connu dans les micro-contrôleurs sans OS : il s'agit du mécanisme d'interruption.

Un système d'exploitation temps réel (*Real-Time Operating System*, RTOS) a pour objectif principal optimiser ses temps de réponse. Il doit permettre au développeur de prédire son comportement et ses temps de réponse, notamment en fournissant des outils adaptés.



Un système d'exploitation non temps-réel (OS grand public) a pour objectif principal de procurer un confort d'utilisation, par exemple en garantissant une certaine réactivité vis-à-vis de l'utilisateur ou en lui proposant un large catalogue de fonctionnalités.

Applications

Exemples : Automobile, médical, aéronautique, spatial, …

Exemples : Lave-linge, machine à café, domotique, ...

→ La criticité des contraintes temporelles dépend de l'application visée !

Ces systèmes peuvent devoir gérer des évènements et/ou des activités en concurrence.

De plus, certaines activités peuvent avoir une criticité plus grande que d'autres. Certaines tâches seront donc prioritaires et feront en sorte que le système préempte l'exécution des tâches de plus faible priorité ;

Temps-réel strict

Un dépassement des contraintes temporelles (réponse trop tardive) est critique, voire catastrophique et entraîne la faute du système.

Ex. : Pilotage automatique d'avion, contrôle d'un réacteur de centrale nucléaire, EPS de voiture, ...

Temps-réel souple

Un dépassement des contraintes temporelles n'est pas critique, mais peu tout de même rendre le système inutilisable. Un dépassement exceptionnel est toléré et pourra être compensé à court terme.

Ex. : visioconférence, machine à café, ...

Temps-réel ferme

Souple pour les fonctionnalités non-critiques du système, dur sinon.

Ex. : dans une voiture, la commande d'injection est critique mais pas le guidage par GPS.

Strict, souple ou ferme ...

→ Le système est catégorisé selon la tolérance accordée à la non-réponse, pas selon son temps de réponse !

Autrement dit, la question à se poser n'est pas « répond-il à temps ? » mais « à quel point le retard est-il grave ? »

→ Un système peut contenir des sous-systèmes, chacun ayant ses propres contraintes temps-réel.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Attention aux amalgames !

~~- Système temps-réel = système rapide~~

→ Le temps-réel garanti que le système est déterministe (répondra en un temps connu).

~~- Programmation temps-réel = assembleur~~

→ La programmation se fait généralement en C (ou dérivés).

~~- Augmentation de la vitesse des processeurs = diminution des problèmes temps-réel~~

→ Cf premier point, la vitesse ne garanti pas le déterminisme d'un système.

→ Si on prend un OS classique sous prétexte qu'on dispose d'un processeur plus performant, toutes les fonctionnalités non maîtrisées de l'OS rendent le système non-déterministe.

MÉCANISMES DES SYSTÈMES D'EXPLOITATION TEMPS-RÉEL

Caractéristiques des RTOS

Ordonnancement / *Scheduling*

Synchronisation et Communication

Mutex

Sémaphore

Queue de messages

ENSI CAEN

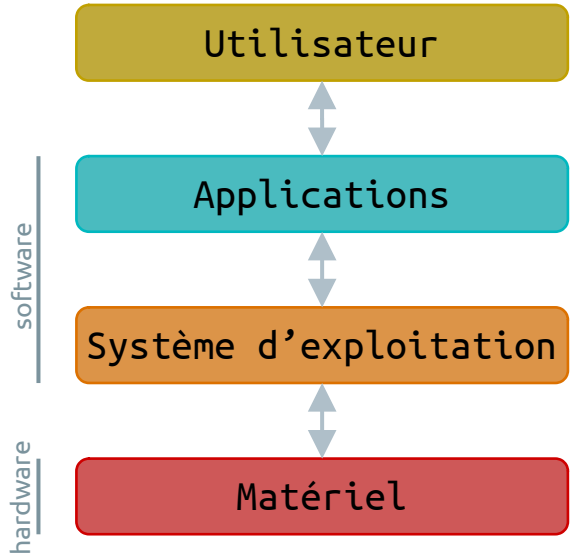
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Un RTOS doit fournir plusieurs services

- Garantie de performances temporelles
  - temps de réponse, temps de préemption, ...
- Facilité d'accès aux ressources physiques
  - temps CPU, mémoire, périphériques, …
- Simplicité de développement

… malgré ses contraintes

- Capacités limitées — puissance de calcul, stockage mémoire, …
- Gestion de l'énergie — consommation/autonomie, échauffement
- Portabilité — sur différents processeurs et/ou couches systèmes



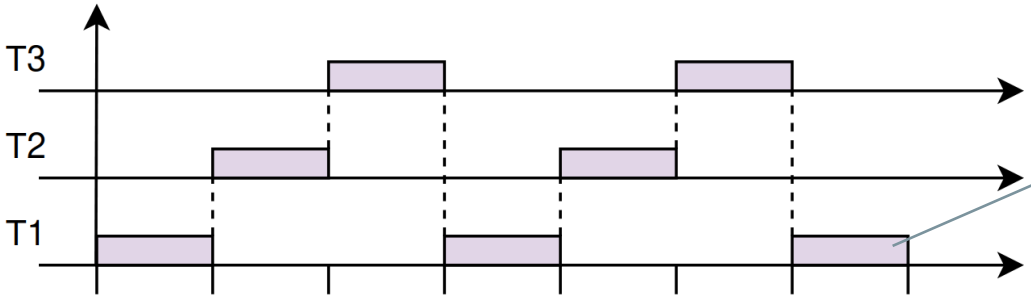
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Les contraintes temps-réel doivent être maîtrisées

Prédictibilité	→ choix de l'architecture hard+soft (cache ?, archi CPU ?, protocoles de comm. ?) *→ aspect abordé à travers d'autres enseignements (SE, archi //, archi calcul, ...)*
Respect des échéances	→ ordonnancement
Compétition des tâches	→ mécanismes de synchronisation (synchrone, asynchrone)
Coopération des tâches	→ mécanismes de communication (sémaphore, mutex, queue de messages)

Un OS multi-tâche doit permettre l'exécution de plusieurs tâches en simultané (en apparence tout du moins).

À l'échelle du processeur, les tâches sont exécutées par fragments temporels, de façon séquentielle :



L'unité de temps de calcul CPU dédié aux fragments des tâches est appelé « slot ».

L'OS est capable de basculer de l'exécution d'une tâche vers une autre. Mais quelle tâche choisir ?

→ Choisir la tâche à exécuter est le rôle de l'**ordonnanceur (*scheduler*)**.

→ L'ordonnanceur est la partie fondamentale des OS multi-tâches.

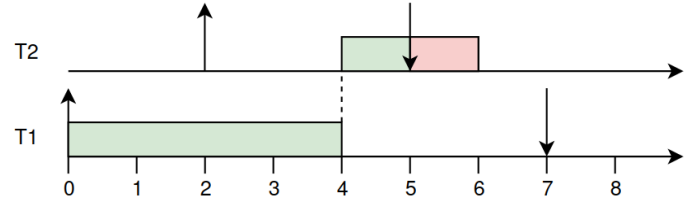
L'ordonnanceur se caractérise par sa politique d'ordonnancement, c-à-d sa méthode de construction de l'emploi du temps du CPU.

Exemple de tâches à gérer :

Tâche	Arrivée	Temps d'exéc.	Échéance
T1	0	4	7
T2	2	2	5

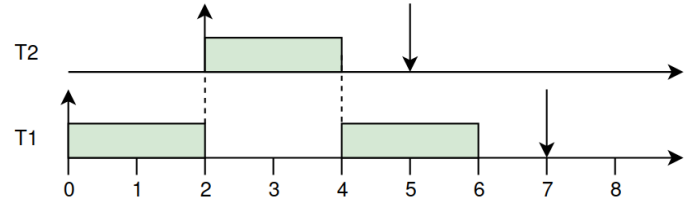
- Politique 1 :

  Premier servi = premier arrivé, sans préemption



- Politique 2 :

  Premier servi = le plus prioritaire, avec préemption



Remarque 1 : il s'agit ici clairement d'un problème d'ordonnancement. Changer le CPU ou optimiser le code ne servirait à rien.

Remarque 2 : si la somme des temps d'exécution est supérieure aux échéances, il s'agit alors d'un problème de ressources et non d'ordonnancement.

Suivant l'instant où l'emploi du temps CPU est décidé, un ordonnanceur peut être classifiée en deux groupes :

Offline

La séquence d'exécution est calculée (et programmée) avant l'exécution effective.

L'ordonnanceur est en réalité un simple séquenceur (ce n'est pas multi-tâches).

→ Pratique pour un faible nombre de tâches connues et périodiques.

Online

Les décisions d'ordonnancement sont prises pendant l'exécution.

→ besoin d'une politique d'ordonnancement (gérée par un algorithme).

Ces ordonnanceurs sont ensuite classifiés en deux groupes : préemptif et non-préemptif.

Exemple : ordonnanceur offline (séquenceur)

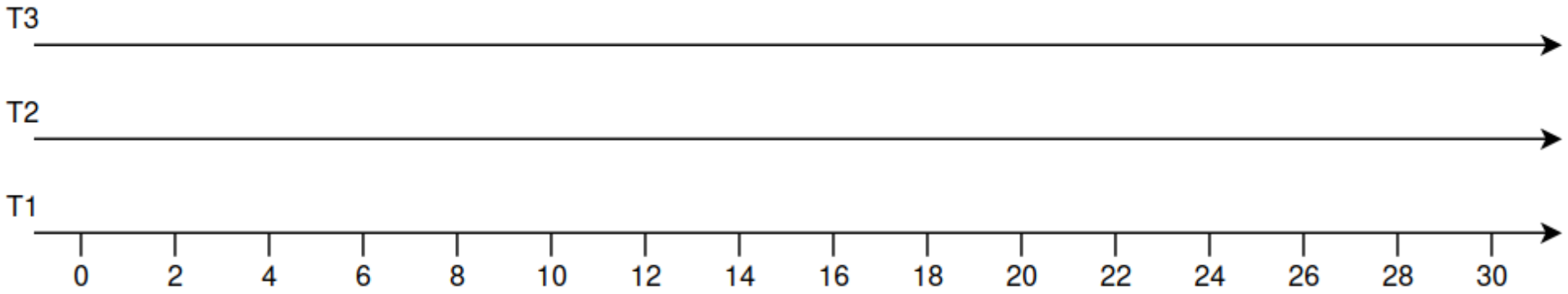
Tâche	Temps d'exécution	Périodicité
T1	2	10
T2	4	15
T3	2	6

L'unité de temps est le temps de cycle CPU.

→ *Time slot* : plus grand diviseur commun des temps d'exécution = ___

→ Cycle majeur : plus petit multiple commun des périodes = ___

Construisons le chronogramme :



Bilan : ordonnanceur offline (séquenceur)

Avantages :

Programmation simple

Ultra-déterministe (la séquence est programmée à la compilation, et non pas gérée à l'exécution=

Très adapté si peu de tâches

Inconvénients :

Il faut tout calculer à l'avance pour programmer correctement

Construction difficile pour tâches nombreuses

Peu évolutif (tout casser pour ajouter une tâche)

Ordonnancement non-préemptif (ou coopératif)

La tâche courante continue de s'exécuter tant qu'elle ne rend pas la main explicitement (via appel système).

Les interruptions ne peuvent pas modifier la configuration d'exécution des tâches !

\+ Très simple à implémenter → permet le multi-tâches sur de petits MCU.

\- Temps-réel mal maîtrisé ;

\- Une tâche mal programmée risque de garder la main durant une longue période ;

\- Les tâches ne peuvent pas être synchronisées avec les interruptions.

Ordonnancement préemptif

Le système a la possibilité d'interrompre la tâche en cours afin de forcer un ré-ordonnancement.

En général, un *timer* est démarré afin de forcer périodiquement un ré-ordonnancement (« **tick** »).

+ Respect des priorité des tâches ;
+ Aspects temps-réel mieux maîtrises ;
+ Tâches implémentées sans ce soucier du ré-ordonnancement.

- Plus complexes à implémenter.

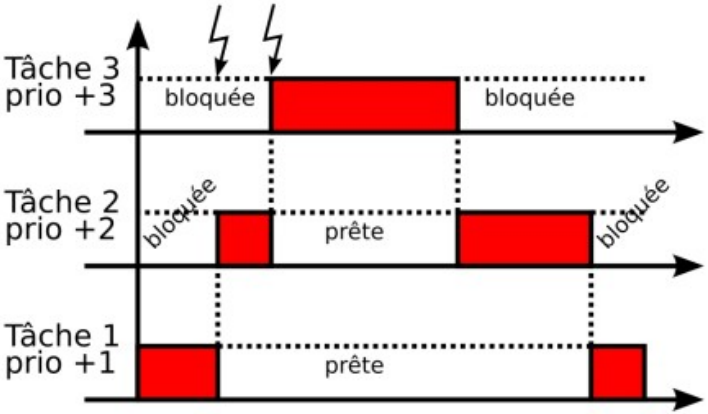
FreeRTOS propose les deux mécanismes (coopératif et préemptif).
Le choix est à définir par l'utilisateur dans le fichier de configuration `FreeRTOSConfig.h`.

Exemples de politiques d'ordonnancement

Par priorité, sans partage de temps

La priorité est le critère absolu de décision. Une tâche de priorité inférieure à la tâche active ne pourra être exécutée que lorsque cette dernière passera à l'état bloqué ou se terminera.

→ Problème : que se passe-t-il si deux tâches on la même priorité ?

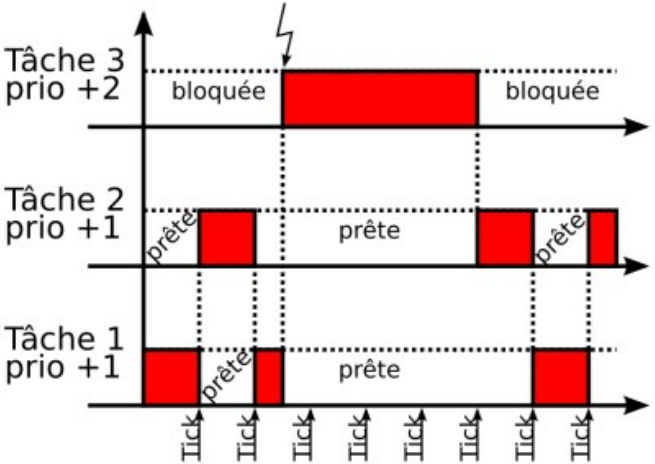


Par priorité, avec partage de temps

Partage de temps équitable entre les tâches de priorité égale : ***Round Robin***.

→ Uniquement sur systèmes préemptifs.

→ C'est le plus rencontré sur les RTOS.



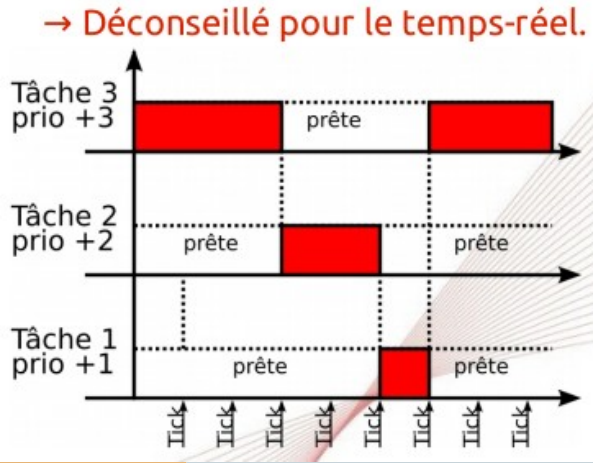
Ré-ordonnancement collaboratif

La priorité n'est plus un critère absolu. La majorité du temps CPU est donnée aux tâches les plus prioritaires, mais une petite partie est affectée aux autres tâches afin qu'elles continuent d'avancer.

→ Uniquement sur systèmes préemptifs.

→ Politiques d'ordonnancement complexes.

→ Déconseillé pour le temps-réel.



États d'une tâche

Afin d'optimiser le traitement des tâches, l'ordonnanceur va leur attribuer un état.

Ces états sont propres à chaque OS, même si certains états sont très communs.

Exemples avec FreeRTOS

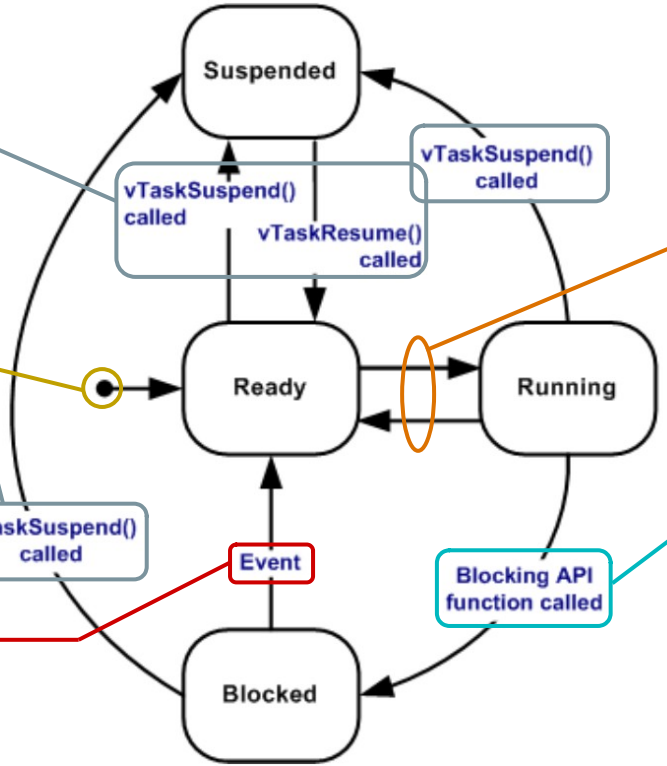
- `Ready` : tâche prête à être exécutée
  - Quand il doit choisir quelle tâche exécuter, l'ordonnanceur choisit parmi les tâches à l'état `Ready`.
- `Running` : tâche en cours d'exécution par le CPU
  - Il y a autant de tâches à l'état `Running` qu'il y a de processeurs disponibles (généralement 1 en systèmes embarqués).
- `Blocked` : non exécutable car en attente d'une ressource externe (signal, données, message, synchro, …)
  - L'ordonnanceur surveille ces tâches pour détecter le moment où elles repassent à l'état `Ready`.
  - Toute tâche qui n'a rien à faire doit être placée dans l'état `Blocked`.
- `Suspended` : la tâche n'est plus vue par l'ordonnanceur.

Transitions entre états avec FreeRTOS

Demande explicite
par le développeur

Création

Ressource disponible
ou évènement produit



Géré par l'ordonnanceur
(il y a quelques exceptions)

Attente d'une ressource
ou d'un évènement

Les fonctions susceptibles de passer une tâche à l'état bloquée sont appelées « fonctions bloquantes ».

https://www.freertos.org/RTOS-task-states.html

Besoins en synchronisation et communication

Le résultat des travaux d'une tâche peuvent être nécessaire pour l'exécution d'un autre tâche.

Ex. : Pour la décompression MPEG2, deux tâches distinctes décodent le son et l'image, une troisième joue le résultat synchronisé.

Des tâches peuvent se partager l'accès à des ressources (données en mémoire, accès matériels).

Ex. : Un navigateur et un client de visio ont tous les deux besoin d'accéder au périphérique réseau.

En communication

Besoin de satisfaire les contraintes d'échanges d'information entre les tâches.

En synchronisation

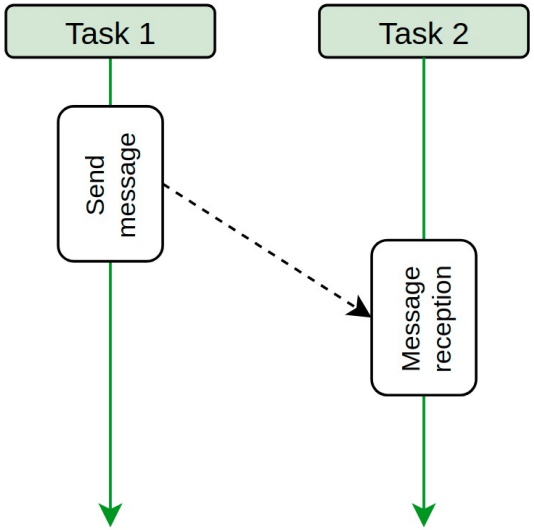
Besoin de satisfaire les contraintes sur l'entrelacement des actions de plusieurs tâches.

Besoin de résoudre les problèmes d'interférences (exclusion mutuelle → mutex)

Concepts fortement liés

Communication : message asynchrone

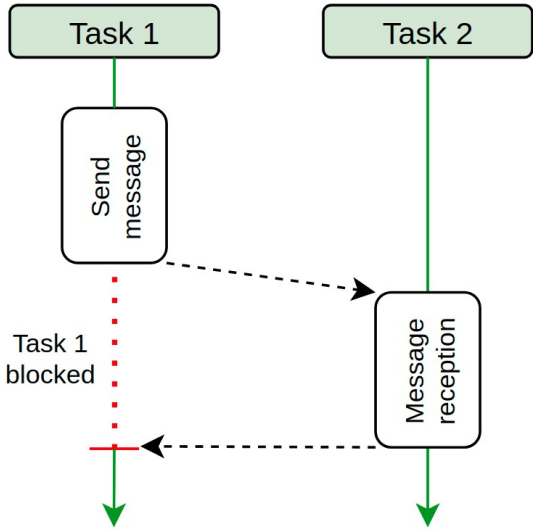
Une tâche peut poster des messages à destination d'une autre tâche, sans soucis de synchronisation.



→ Nécessite un *buffer* pour stocker les messages en attendant leur lecture.

Communication : message synchrone

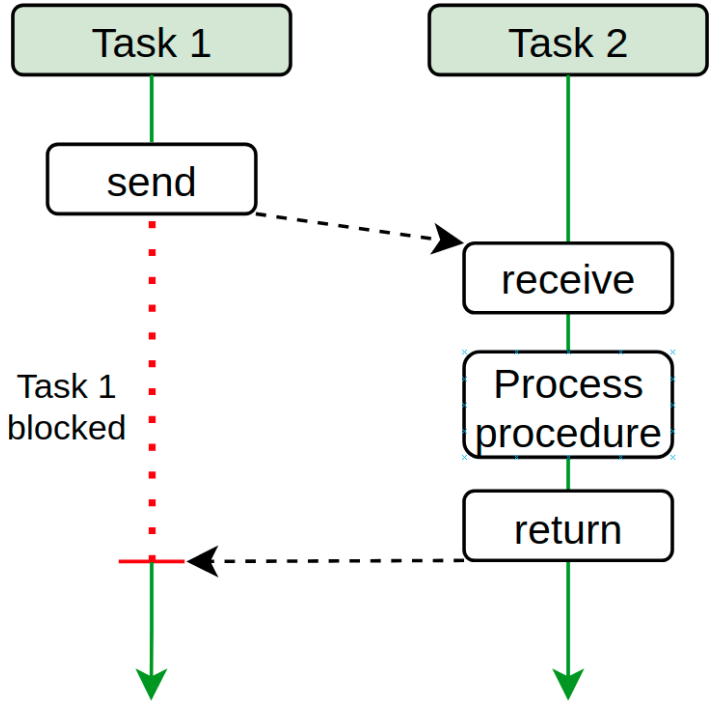
Une tâche poste des messages à destination d'une autre tâche de manière synchrone.



→ Pas besoin de *buffer* , la tâche émettrice attend que la tâche réceptrice ait reçu le message

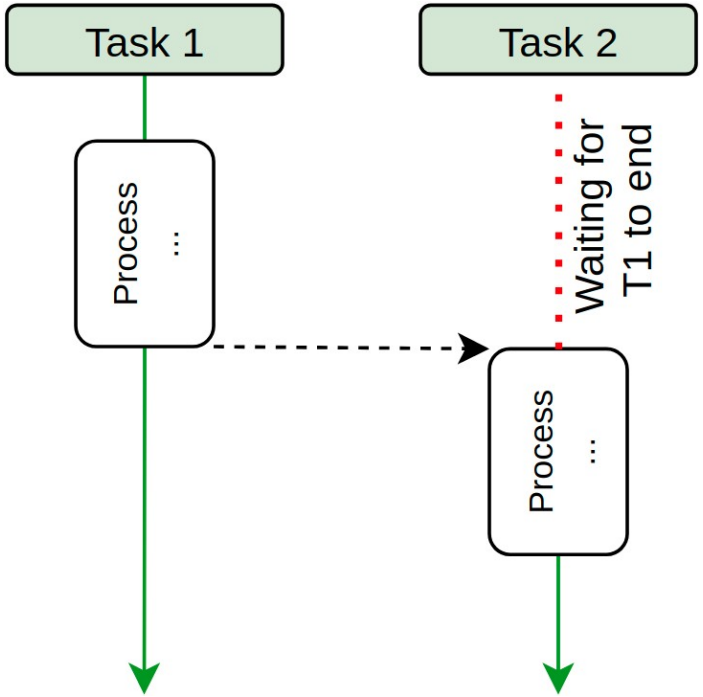
Communication : appel de procédure distant (*Remote Procedure Call*, RPC)

Une tâche appelle une autre tâche pour lui déléguer un traitement, généralement via un réseau.



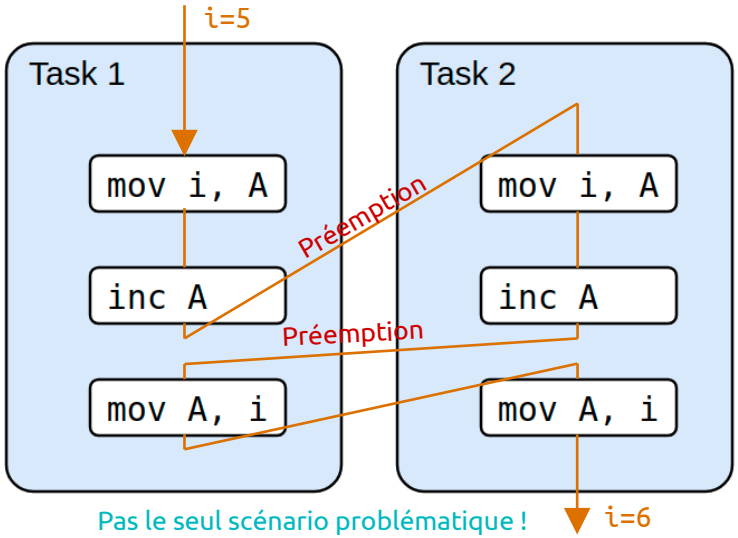
Synchronisation : attente conditionnelle

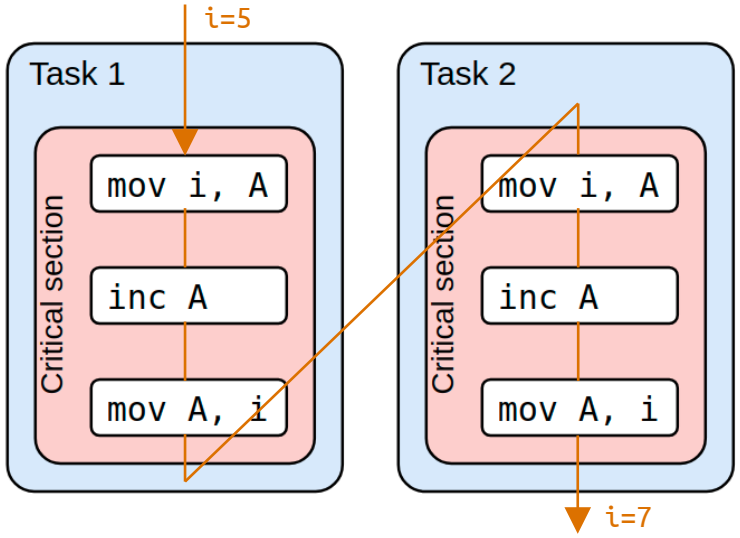
Une tâche doit attendre la fin d'une autre tâche pour exécuter son action.



Synchronisation : problème d'interférence (ressources partagées)

Ex. : deux tâches veulent incrémenter la même variable partagée i (en asm : *load*, *modify*, *store*).





→ Il faut garantir l'indivisibilité des séquences d'instructions pour les sections critiques

→ Mécanisme d'exclusion mutuelle

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Synchronisation : problème d'interférence (ressources partagées)

Solution 1 : Algorithme de Peterson (1981)

```
// Task 1 main routine
while(1) {

    ...
    /* NON-CRITICAL SECTION */
    let_task1_in = true;    // T1 ask for access
    task_to_exe  = 2;       // T1 let T2 going in first

    while( (let_task2_in == true) && (task_to_exe == 2) ) ;

    /* START OF CRITICAL SECTION */
    ...
    ...

    /* END OF CRITICAL SECTION */
    let_task1_in = false;
    ...
}
```

```
// Task 2 main routine
while(1) {

    ...
    /* NON-CRITICAL SECTION */
    let_task2_in = true;    // T2 ask for access
    task_to_exe  = 1;       // T2 let T1 going in first

    while( (let_task1_in == true) && (task_to_exe == 1) ) ;

    /* START OF CRITICAL SECTION */
    ...
    ...

    /* END OF CRITICAL SECTION */
    let_task2_in = false;
    ...
}
```

En pratique, les variables `let_taskN_in` sont rassemblées sous un tableau de booléens (1 case par tâche concurrente).

Problème : Attente active, la tâche ne peut pas être bloquée !

Synchronisation : problème d'interférence (ressources partagées)

Solution 2 : instruction **atomique** `test_and_set()` ⇔ `if(b==0) {b=1 ; return 0;} else{ return 1; }`

```
// Task 1 main routine
while(1) {

    ...
    /* NON-CRITICAL SECTION */
    while( test_and_set(flag) ) ;

    /* START OF CRITICAL SECTION */
    ...
    flag = 0 ;

    /* END OF CRITICAL SECTION */
    ...
}
```

```
// Task 2 main routine
while(1) {

    ...
    /* NON-CRITICAL SECTION */
    while( test_and_set(flag) ) ;

    /* START OF CRITICAL SECTION */
    ...
    flag = 0 ;

    /* END OF CRITICAL SECTION */
    ...
}
```

Instruction atomique = instruction indivisible, ne peut matériellement pas être interrompue.

→ chaque processeur a son propre lot d'instructions atomiques.

Problème : Attente active, la tâche ne peut pas être bloquée !

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Synchronisation : problème d'interférence (ressources partagées)

Solution 3 : utiliser les outils proposés par les systèmes d'exploitation

- Queue de messages → Communication
- Sémaphore → Synchronisation
- Mutex binaire
- Mutex à compteur

→ Partage de ressource

→ Permet de placer les tâches à l'état bloqué, éliminant ainsi le problème d'attente active.

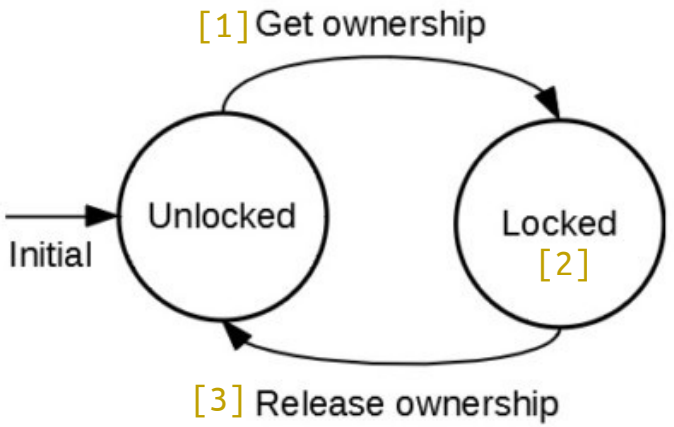
Mutex (*Mutal Exclusion*) : **mécanisme de verrouillage des ressources partagées**

Un mutex est un jeton (une clé) qui contrôle l'accès à une ressource partagée, ou à une section critique de code.

Avant de pouvoir accéder à la dite ressource partagée, une tâche doit d'abord obtenir ce jeton [1]. Une fois ce jeton en sa possession, elle empêche toute autre tâche d'accéder à la ressource protégée [2].

En ce sens, la tâche devient propriétaire du mutex : seule cette tâche peut décider de rendre le mutex [3] (et donc l'accès aux ressources partagées).

Une tâche peut se charger de créer/supprimer un mutex, peut demander à acquérir le mutex et peut le restituer.

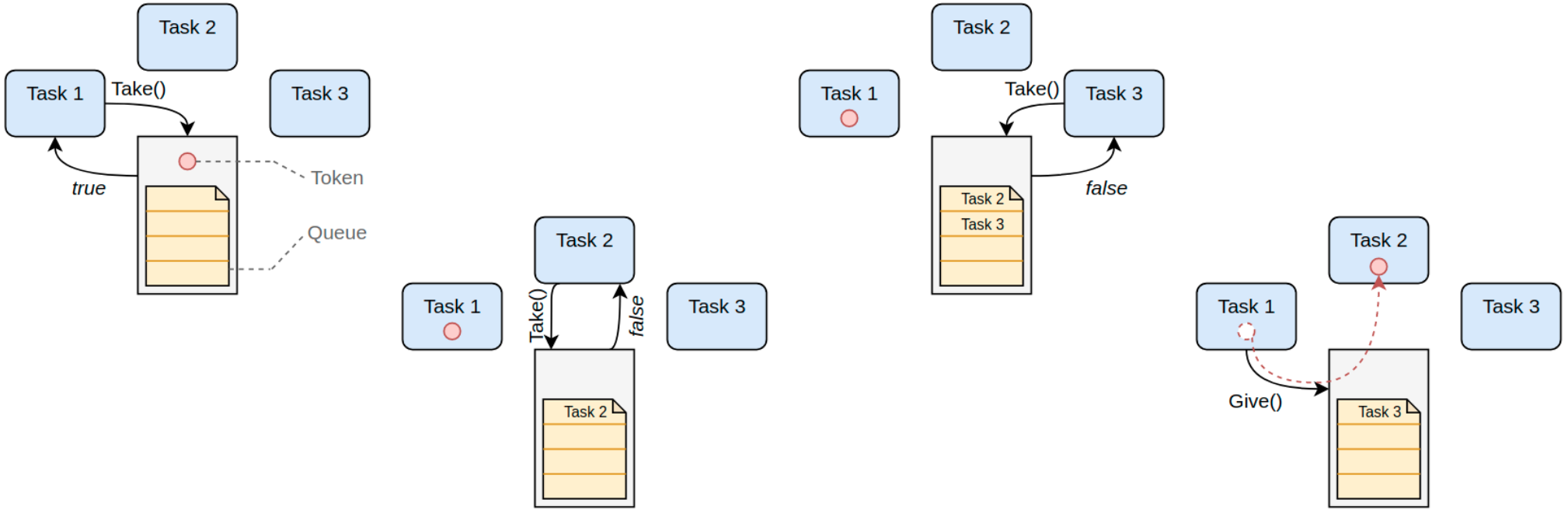


Ex. : Deux particuliers (deux tâches) veulent louer une voiture (ressource), le seul qui peut l'utiliser est celui qui a la clé (mutex). L'autre personne (tâche) doit attendre que la clé (mutex) soit remise à sa place, ce qui signifie que la voiture (la ressource) est libre.

Mutex

Un mutex est implémenté par :

- une variable booléenne (qui indique si le mutex est libre ou non) ;
- une file d'attente : une tâche n'ayant pas obtenu l'accès au mutex se place dans le file d'attente et passe en attente passive.



Doc FreeRTOS : https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html

Problème : inversion de priorité

Si une tâche de faible priorité est propriétaire d'un mutex et qu'une tâche de priorité haute demande ce mutex, alors la tâche de priorité haute sera bloquée tant que la tâche de priorité basse n'aura pas rendu le mutex.

Cette situation s'appelle inversion de priorité, puisque la tâche devient artificiellement prioritaire par rapport à l'autre.

Situation encore pire : un tâche de priorité intermédiaire peut bloquer l'exécution de la tâche de priorité faible, qui mettra encore plus de temps à rendre le mutex pour permettre l'exécution de la tâche de priorité haute.

Solutions

Des protections internes au RTOS (plus précisément au kernel) permettent généralement d'élever temporairement la priorité de la tâche détenant le mutex, afin qu'elle libère le mutex le plus rapidement possible.

Mécanisme d'héritage de priorité (PIP) : la tâche détenant le mutex est automatiquement mise à une priorité supérieure à n'importe quelle autre tâche demandant le même mutex. VOIR CHAPITRE PLUS LOIN.

Mécanisme de plafond de priorité (PCP) : chaque tâche est crée avec une priorité et une priorité maximale atteignable. Chaque tâche qui acquiert un mutex passe à sa valeur de priorité maximale. VOIR CHAPITRE PLUS LOIN.

Mutex à compteur

Utilise les mêmes API de manipulation que les mutex binaires. La file d'attente est conservée, mais la variable booléenne est remplacée par une variable entière.

Principe

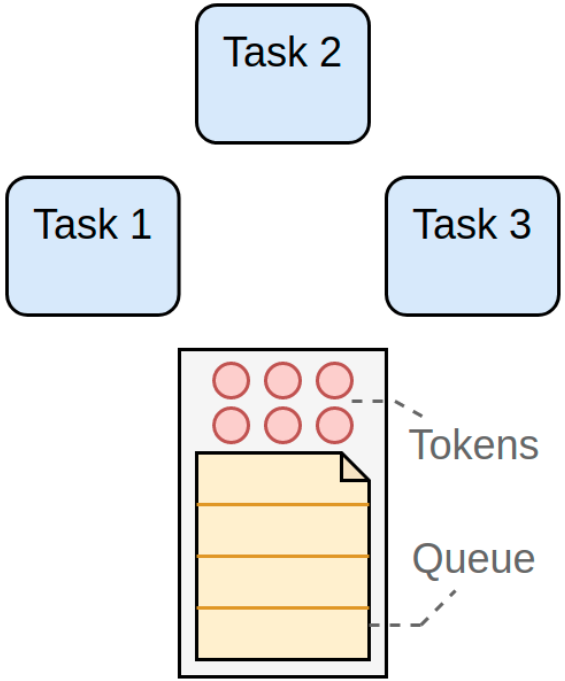
Une variable entière `n` est initialisée au nombre maximal d'accès simultanés à la ressource.

Si le mutex est disponible (`n > 0`), alors une tâche peut en devenir co-propriétaire.
La variable `n` est décrémentée et la tâche poursuit son traitement.

Si le mutex n'est pas disponible (`n = 0`), la tâche se bloque et se met en file d'attente.

Lorsqu'une tâche libère le mutex, la variable `n` est incrémentée.

Ex. : utilisé dans les modules DMA (Direct Memory Access) à plusieurs canaux. Tant qu'un canal est libre, une tâche peut l'utiliser.



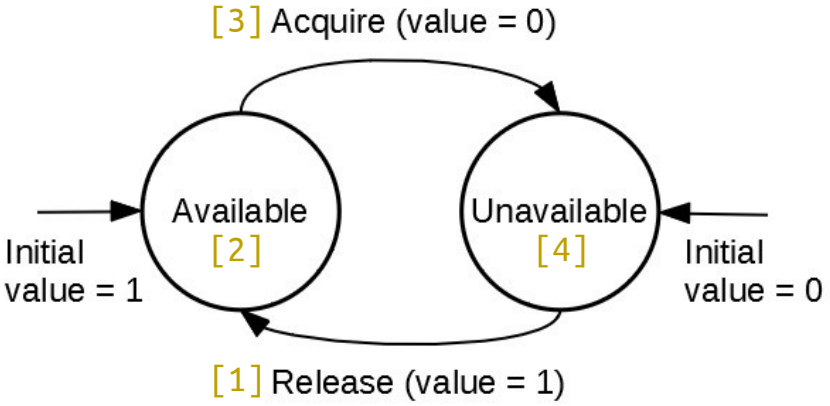
Sémaphore binaire

Mécanisme de signalisation (≈ *flag*), généralement utilisé pour la synchronisation de tâches.

En passant le sémaphore à '1' [1], une tâche indique qu'elle est arrivée au lieu de rendez-vous (ou qu'un résultat est disponible) [2].

Une autre tâche acquittera le rendez-vous en passant la valeur du sémaphore à '0' [3].

Le sémaphore est alors dans un état indique qu'aucune tâche n'est arrivé au rendez-vous (ou qu'aucun résultat n'est disponible) [4].

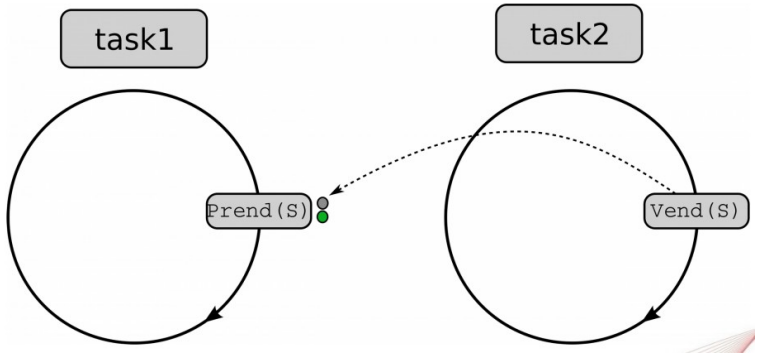


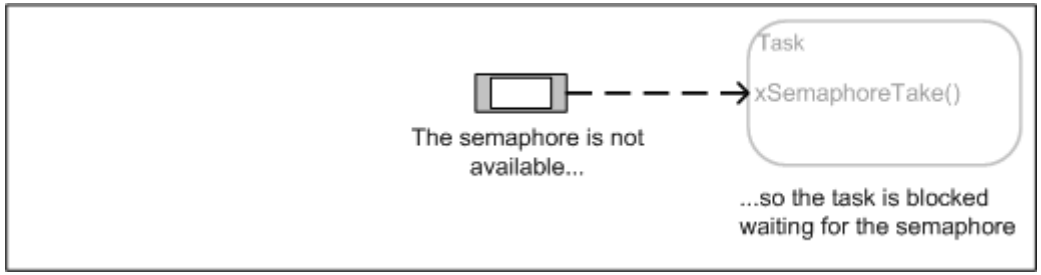
Contrairement au mutex, aucune tâche n'est propriétaire d'un sémaphore : n'importe quelle tâche peut signaler sa présence (ou la présence de résultat) [1] et n'importe quelle tâche peut accepter le rendez-vous ou récupérer le résultat [3].

Sémaphore binaire

Le sémaphore binaire est implémenté par :

- une variable booléenne (qui indique si le sémaphore est libre ou non) ;
- une file d'attente : une tâche n'ayant pas obtenu l'accès au sémaphore se range dans la file d'attente et passe en attente passive.





Ex. : La tâche 1 attend le résultat de l'ADC pour l'afficher sur un écran. La tâche 2 (ISR de l'ADC) mettra le sémaphore à '1' quand le résultat sera acquis. La tâche 1 d'affichage patiente (bloquée, en attente passive) tant que le sémaphore n'est pas mis à '1' par la tâche 2, puis reprendra son traitement et repassera le sémaphore à '0' pour indiquer que le résultat a été récupéré.

Doc FreeRTOS : https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html

Sémaphore binaire vs. Mutex

Ces deux mécanismes sont *a priori* très semblables, mais attention à ne pas les confondre.

Sémaphore binaire

- Rôle : synchronisation de tâches.

- Pas de propriétaire : les actions de prise et de restitution du sémaphore ne sont pas effectuées par la même tâche.

- Inversion de priorité : l'utilisation de sémaphore ne crée pas de situation d'inversion de priorité.

- ISR : très utilisé dans les ISR pour passer le moins de temps possible dans ces routines, tout en signalant qu'un évènement s'est produit.

Mutex

- Rôle : protection de ressources partagées.

- Tâche propriétaire : un tâche détient le mutex et seule celle-ci peut le restituer.

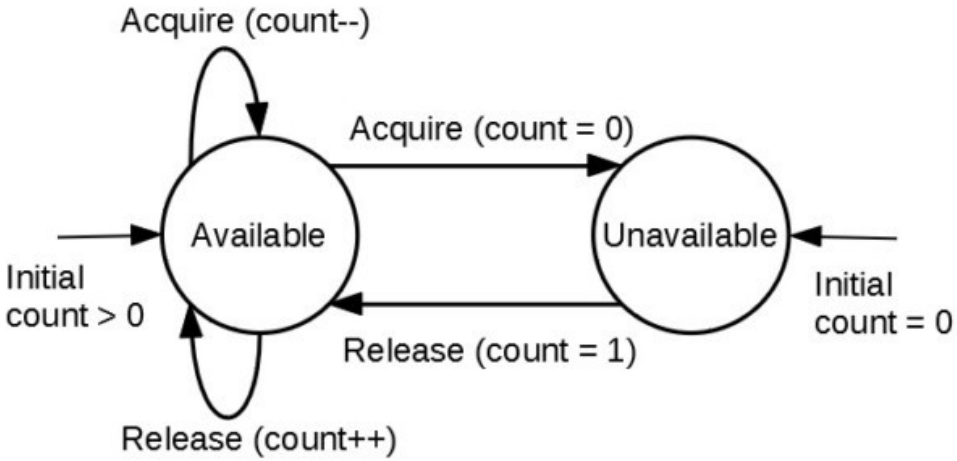
- Inversion de priorité : le RTOS possède un mécanisme de protection pour minimiser les effets de cette situation.

- ISR : fortement déconseillé car risque de bloquer une tâche dans une ISR (comportement fortement non temps-réel).

Sémaphore à compteur

Comportement analogue au sémaphore binaire : la file d'attente est conservée, mais la variable booléenne est remplacée par une variable entière.

Peut permettre de compter un nombre d'occurrences d'un évènement (en incrémentant le compteur) et le nombre de fois où l'évènement a été traité (en décrémentant le compteur).



Exercice

Une application nécessite que ses trois tâches soient exécutées dans l'ordre : T3, T1, T2, T3, T1, T2, T3, …

Q1. Proposez une solution permettant de faire cela en utilisant 3 sémaphores binaires (S1, S2, S3).

On précise que T1 doit obtenir S1 pour s'exécuter, T2 doit obtenir S2, …

On appellera `Give(Sx)` et `Take(Sx)` les fonctions permettant respectivement de donner ou prendre le sémaphore Sx.

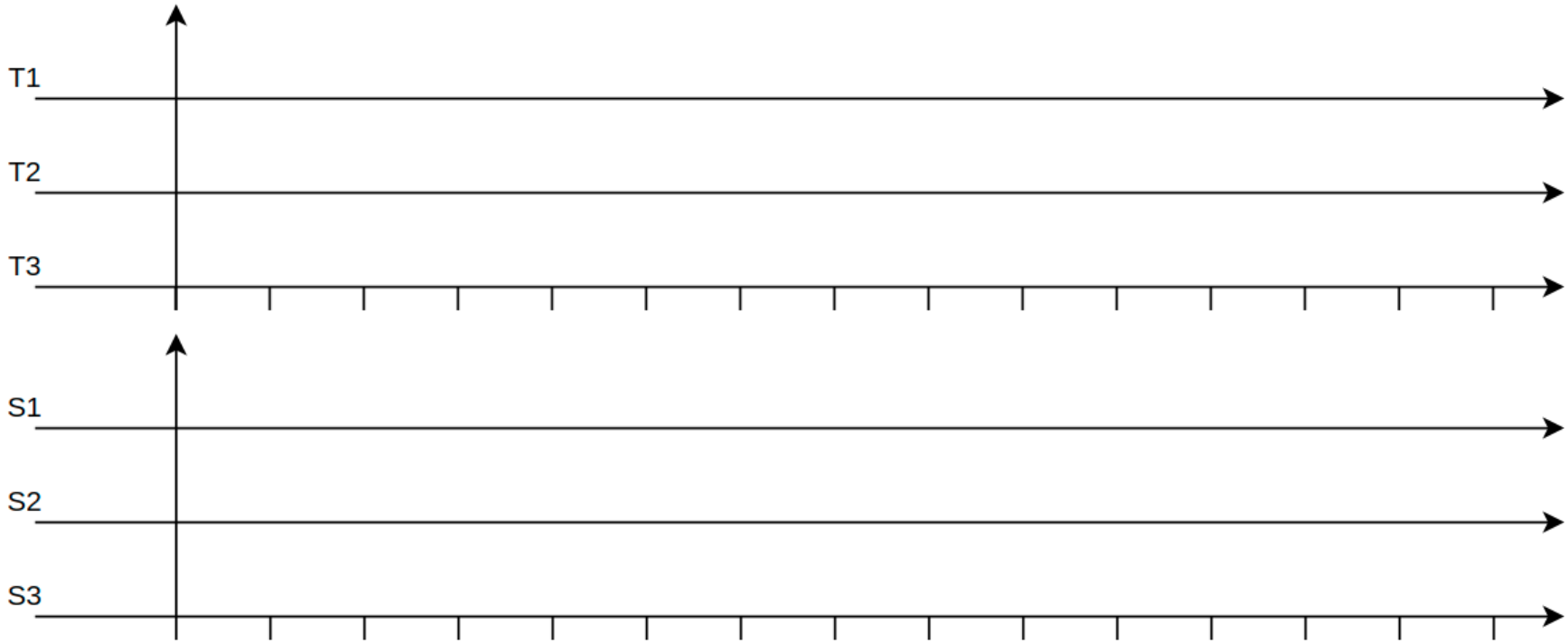
```
void Task1(void *pvParam){

  while(1){
    ...
    ...
    ...
  }
}
```

```
void Task2(void *pvParam){

  while(1){
    ...
    ...
    ...
  }
}
```

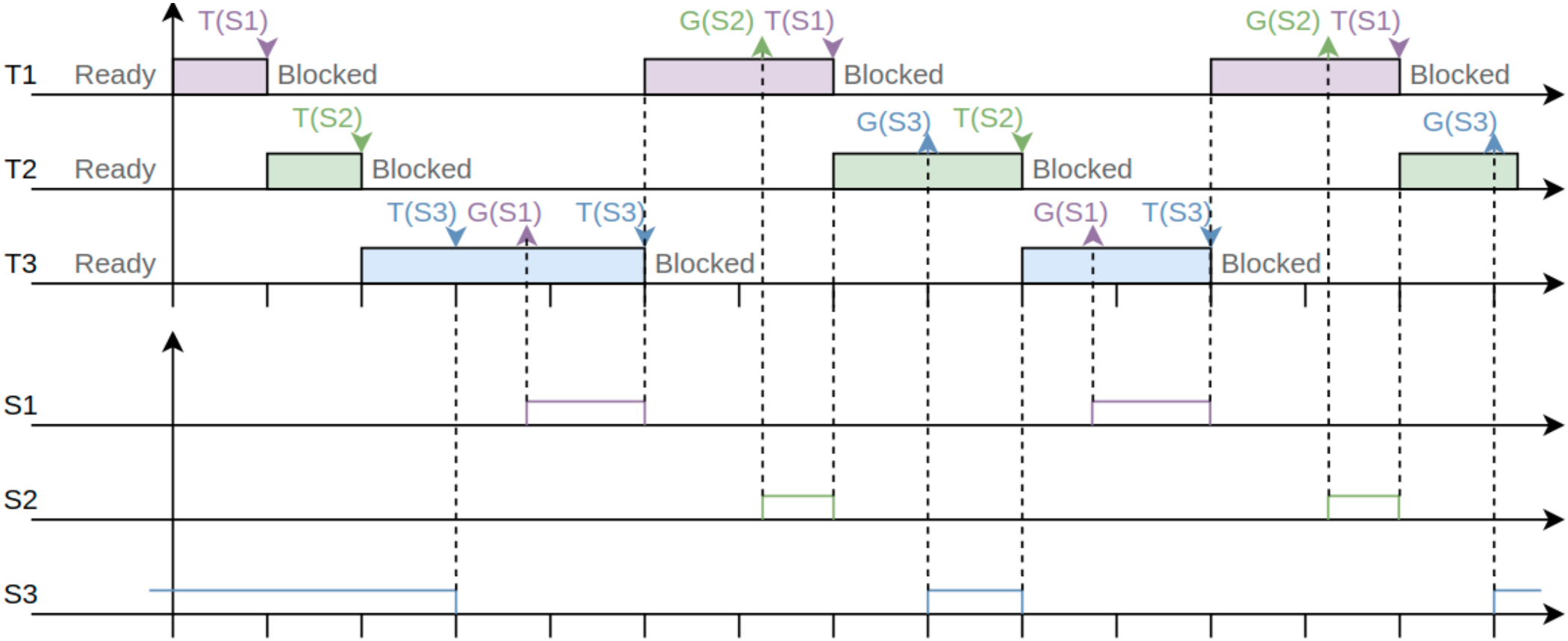
```
void Task3(void *pvParam){

  while(1){
    ...
    ...
    ...
  }
}
```

Exercice

Q2. Complétez le chronogramme page suivante en indiquant la totalité des mouvements de sémaphores (prise et restitution) et l'état de chaque tâche (Ready, Running, Blocked).



Sémaphore


T1
Ready
T(S1)
Blocked
G(S2) T(S1)
Blocked
G(S2) T(S1)
Blocked
T2
Ready
T(S2)
Blocked
G(S3) T(S2)
Blocked
G(S3)
T3
Ready
T(S3) G(S1) T(S3)
Blocked
G(S1) T(S3)
Blocked
S1
S2
S3

Queue de messages (message queues)

Permet l'échange de données (communication) entre tâches. Construit avec deux files d'attente.

File d'attente des données

Contient un nombre fini d'éléments. Le nombre d'éléments et leur taille sont configurables.

File généralement gérée comme une pile FIFO (First In = First Out), mais pas seulement.

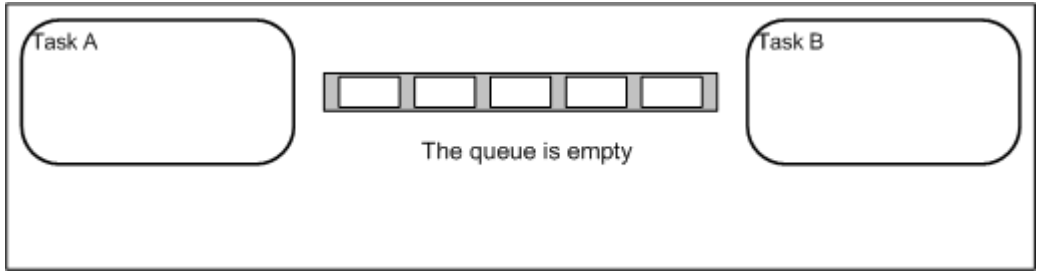
File d'attente des tâches

Les tâches qui demandent la lecture d'un message alors que la queue est vide viennent dans cette file et se bloquent (attente passive).

La file peut classer les tâches par priorité ou les ranger dans une FIFO.

Exemple

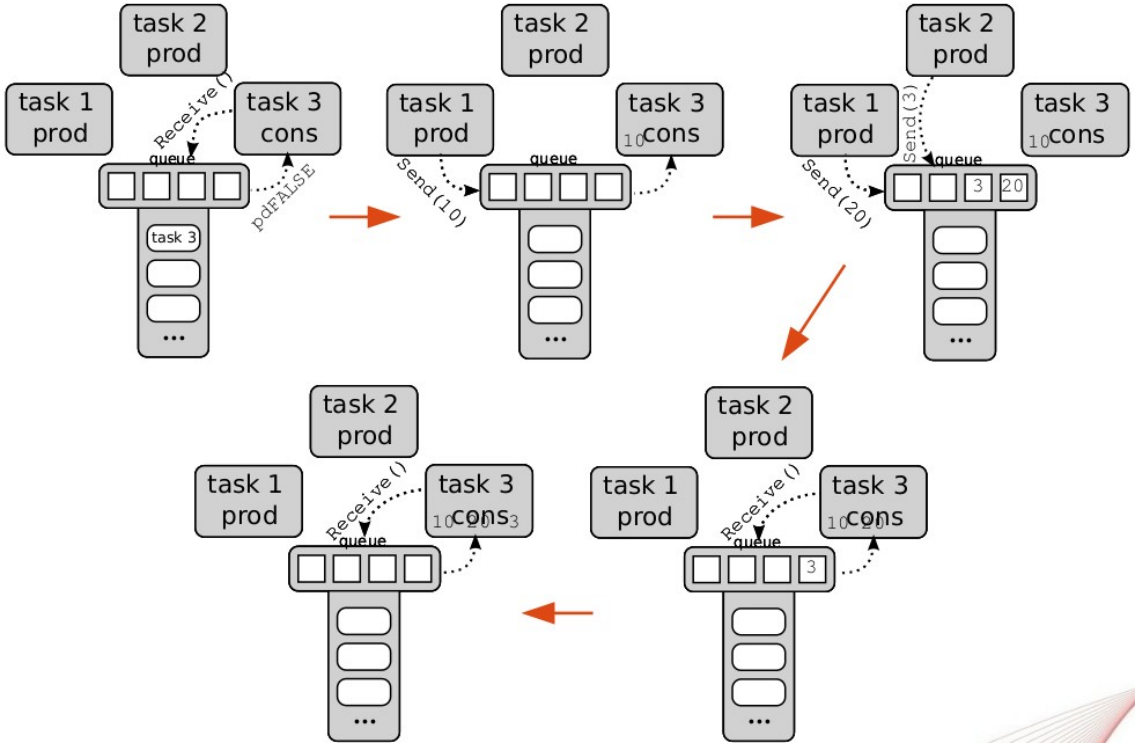
Cas avec de un producteur unique et un consommateur unique.



Doc FreeRTOS : https://www.freertos.org/Embedded-RTOS-Queues.html

Exemple

Cas avec de multiples producteurs et un consommateur unique.



Exercice

Soit une application possédant deux tâches productrices de priorité 1 et une tâche consommatrice de priorité 2, et fonctionnant avec un *kernel* en mode préemptif.

Le code des trois tâches est donné ici.

```
void prodTask1(void *pvParam){
  uint32_t y=1;

  while(1){
    osMessagePut(msgBox, y, 0);
    y+=2;
    osDelay(10);
  }
}
```

```
void prodTask2(void *pvParam){
  uint32_t y=2;

  while(1){
    osMessagePut(msgBox, y, 0);
    y+=2;
    osDelay(10);
  }
}
```

```
void consTask(void *pvParam){
  uint32_t z;
  uint8_t sBuffer[20];

  while(1){
    osMessageGet(msgBox,&z,osWaitForEver);
    sprintf(sBuffer,"%d\r\n",z);
    uartSendString(sBuffer);
  }
}
```

Exercice

Q1. Donnez le chronogramme représentant la séquence d'exécution à partir du démarrage de l'application. Faîtes bien apparaître les instants clés, en indiquant les appels de fonctions associées et l'état des tâches.

Exercice

Q2. Qu'observerait-on au niveau d'un terminal UART branché au système ?

PROGRAMMATION DES RTOS, APPLICATION À FREERTOS

Task Control Block (TCB)

Gestion de la mémoire

Gestion des interruptions

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Task Control Block (TCB)

Pour organiser le planning d'exécution des tâches, l'ordonnanceur doit disposer de nombreuses informations. Pour cela, chaque tâche possède son jeu personnel d'attributs, stockés dans le Task Control Block (TCB).

Ces attributs sont propres à chaque OS, voyons les plus communs ici :

- Adresse de début de code : adresse du début du fil de code à exécuter. Généralement passé sous forme de pointeur de fonction.

- Adresse de début et taille de la pile : chaque tâche possède sa propre pile (*stack*) où sont stockées ses données (variables dynamiques, paramètres de la fonction, adresse de retour).

- Priorité de la tâche : permet à l'ordonnanceur d'effectuer son choix lorsque plusieurs tâches sont « ready ».

- État de la tâche (*) : état courant de la tâche (prête, exécutée, bloquée, …)

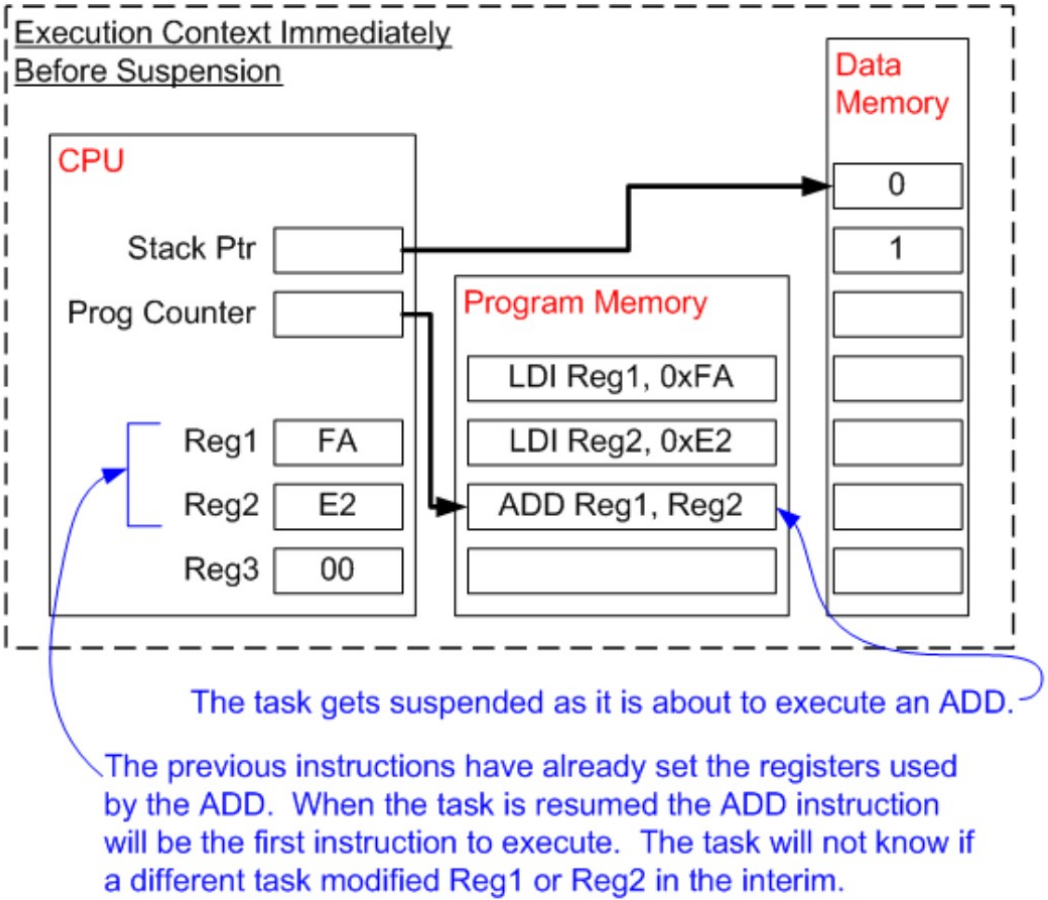
- Ressources bloquantes (*) : si la tâche est à l'état bloquée, indique la raison du blocage (sémaphore, *timer*, …). Permet à l'ordonnanceur de déterminer quelle tâche débloquer lors de la restitution d'une ressource.

(*) → Dans FreeRTOS, ces paramètres ne sont pas stockés dans le TCB de la tâche. À la place FreeRTOS possède ses propres tables contenant la liste des tâches, de leur état et de leur(s) ressource(s) bloquante(s).

Task Control Block (TCB)

Il faut garder à l'esprit que l'ordonnanceur peut à tout moment préempter une tâche en cours d'exécution, sans prévenir !

Lors de la préemption de la tâche, il faut sauvegarder tout son contexte d'exécution (registres CPU, instruction en cours, état de la *stack*, …).



https://www.freertos.org/implementation/a00006.html

Task Control Block (TCB)

Pour faciliter la sauvegarde du contexte, certains attributs du TCB sont associés au contexte d'exécution.

Une fois encore, ces attributs sont propres à chaque OS. Voici quelques attributs génériques :

- Adresse courante de code :

nécessaire pour reprendre la tâche à l'endroit où elle a été interrompue. Cela consiste à sauvegarder la valeur du registre *Program Counter (PC)* du CPU au moment où la tâche est préemptée.

- Adresse courante de la pile :

nécessaire pour replacer le pointeur de pile à l'endroit où il était au moment où la tâche a été interrompue. Cela consiste à sauvegarder la valeur du registre *Stack Pointer (SP)* au moment où la tâche est préemptée.

- Registres du CPU :

l'exécution d'un fil de code utilise des registres du processeur. Lorsqu'une tâche est préemptée, il est nécessaire de sauvegarder la valeur de ces registres puis de les restaurer lorsque la tâche repasse à l'état actif.

Dans FreeRTOS, ces paramètres ne sont pas stockés dans le TCB de la tâche, mais sur la pile de la tâche.

Rappels sur l'allocation mémoire en C

```
char global_var = 5;

void foo(char arg1, char arg2) {
    static char static_var = 0;
    char local_var  = 0;
    return;
}

void main() {
    foo( '1' , '0' ) ;
    char* p = malloc( 15*sizeof(char) );
}
```

- Variable globale → allocation statique
- Variable statique locale → allocation statique
- Variable locale → allocation automatique (sur la pile)
- Argument de fonction → allocation automatique (sur la pile)
- Adresse de retour de fonction → allocation automatique (sur la pile)
- malloc() et free() → allocation dynamique (sur le tas)

Attention : La gestion de la mémoire requiert une très grande rigueur de programmation !

Note : l'allocation dynamique peut ne pas être supportée par l'OS ou par le processeur (si absence de MMU).

Allocation automatique

Chaque tâche possède sa propre pile (*stack*)

Problème : le pointeur de pile (SP) est initialisé lors du *startup* puis, au cours de l'exécution, il est géré automatique par le matériel (appel de fonction, ISR, etc.) et par la *toolchain* (allocation, push, pop, etc.).

Solution : FreeRTOS « trompe » les mécanismes automatiques en modifiant la valeur du SP lors de la restauration d'une tâche : SP = TopStackTaskToSpawn (valeur sauvegardée lorsque la tâche a été préemptée).

La taille de pile d'une tâche est définie lors de sa création

Problème : attention aux débordements de piles (stackOverFlow) !

Solution ? : FreeRTOS ne propose que des mécanismes ultra-rudimentaires de protection de la mémoire. En comparaison, les autres systèmes vont utiliser une MMU (ex : Linux et la fameuse « *segmentation fault* »).

https://freertos.org/Stacks-and-stack-overflow-checking.html (Kernel > Developer Docs > Stack Overflow Protection)

Allocation dynamique

Quand une tâche (ou sémaphore, mailbox, ...) est créée, FreeRTOS doit allouer dynamiquement de la mémoire.

Problèmes :

- `malloc()` et `free()` possiblement non implémentés sur le système (si absence de MMU)
- utilise une grande quantité de mémoire
- un thread peut accéder à la zone allouée par/pour un autre thread
- code non déterministe

Solution :

FreeRTOS implémente ses propres méthodes de gestion du tas : `pvPortMalloc()` et `vPortFree()`

Allocation dynamique

FreeRTOS propose plusieurs stratégies de gestion du tas (*heap*).

Le développeur choisit la stratégie qui lui convient en incluant le fichier source correspondant dans son projet :

- heap_1.c – the very simplest, does not permit memory to be freed. → *Couvre la plupart des applis avec FreeRTOS.*
- heap_2.c – permits memory to be freed, but does not coalescence adjacent free blocks.
- heap_3.c – simply wraps the standard `malloc()` and `free()` for thread safety.
- heap_4.c – coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
- heap_5.c – as per `heap_4.c`, with the ability to span the heap across multiple non-adjacent memory areas

NB : FreeRTOS implémente certaines fonctions en deux versions : pour un usage en statique ou en dynamique.

Ex. : `xTaskCreate()` et `xTaskCreateStatic()` ; `xSemaphoreCreateBinary()` et `xSemaphoreCreateBinaryStatic()` ; …

https://freertos.org/a00111.html (Kernel > Developer Docs > Heap Memory Management)

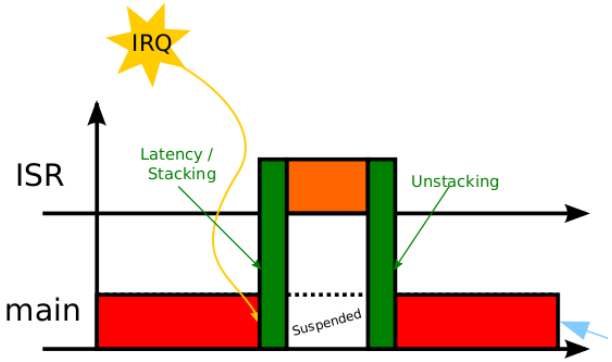
https://freertos.org/Static_Vs_Dynamic_Memory_Allocation.html (Kernel > Developer Docs > Static Vs Dynamic Memory)

Rappels sur les interruptions

Lorsqu'un évènement externe (généralement asynchrone) se produit, il peut nécessiter un traitement dédié.

Le périphérique chargé de capter l'évènement demande une interruption au CPU (IRQ, *Interrupt Request*).

Si le CPU est sensible aux IRQ en général et celle-ci en particulier, alors il interrompt son fil pour exécuter la routine d'interruption correspondante (ISR, *Interrupt Service Routine*).



C'est une manière rudimentaire de transformer un programme mono-tâche en programme multi-tâche sans nécessité d'implémenter un OS !

Attention : il s'agit d'un mécanisme purement **matériel**, qui ne peut pas directement être géré par FreeRTOS !

Interruptions différées (*deferred interrupt processing*)

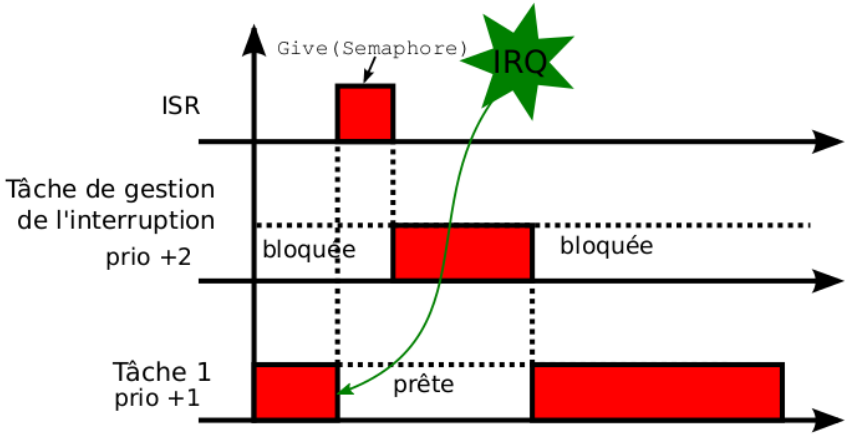
Problème : Une ISR est une routine exécutée par le processeur, hors de contrôle du noyau de l'OS

→ comportement inattendu du noyau ;

→ temps-réel non garanti

Solution : Réduire au maximum le temps d'exécution des ISR

→ utilisation des services fournis par l'OS (sémaphore, mailbox) pour que l'ISR signale le besoin de traitement puis traitement effectif de l'évènement dans une simple tâche



Avec ce système, peu de temps passé dans l'ISR, puisque le traitement de l'interruption est différé.

https://freertos.org/deferred_interrupt_processing.html
(Kernel > Secondary Docs > Deferred Interrupt Handling)

Gestion des interruptions

FreeRTOS laisse au développeur le choix de sa stratégie de gestion des interruptions :

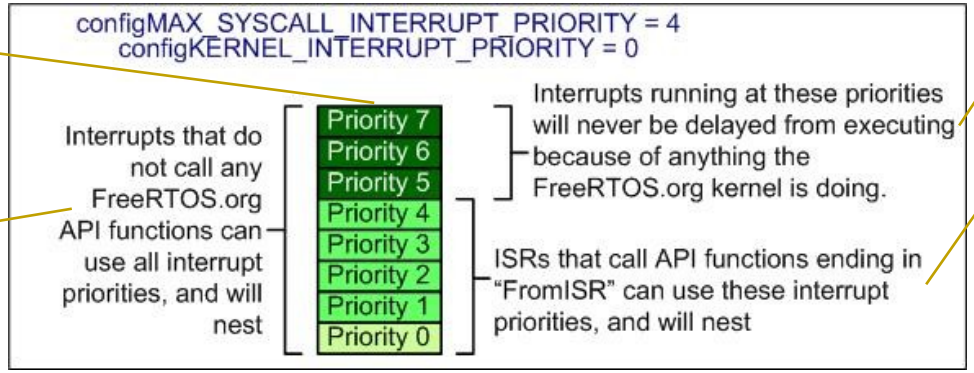
- ISR matérielles (classiques) : mais besoin d'une grande rigueur pour le temps d'exécution des ISR
- ISR différées via les outils de l'OS

Cette stratégie est ajustable avec deux constantes prédéfinies du fichier `FreeRTOSConfig.h`

- `configKERNEL_INTERRUPT_PRIORITY` : niveau de priorité du *kernel* (tick de l'OS)
- `configMAX_SYSCALL_INTERRUPT_PRIORITY` : niveau max de priorité pour lequel une ISR peut utiliser les fonctions de l'API FreeRTOS

Le nombre de niveau de priorité dépend de l'architecture sur laquelle FreeRTOS est déployé.

Une ISR qui n'utilise pas les fonctions de l'API peut avoir n'importe quel niveau de priorité.



Example interrupt priority configuration

Une ISR de plus haute priorité ne sera pas perturbée par l'OS mais ne peut pas utiliser les fonctions de l'API.

Une ISR qui utilise les fonctions de l'API doit être au niveau de priorité ≤ configMAX_SYSCALL_INTERRUPT_PRIORITY si cette constante est implémentée,

ou de niveau de priorité = `configKERNEL_INTERRUPT_PRIORITY` sinon.

https://freertos.org/a00110.html#kernel_priority

TOUR D'HORIZON DES ALGORITHMES D'ORDONNANCEMENT

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

```
/* Section en cours de construction */
```

En attendant, merci à Basile Dufay (Maître de Conférence à l'ESIX de Caen) pour ces diapositives généreusement rendues disponibles pour l'ENSI !

basile.dufay@unicaen.fr

CHOISIR SON SYSTÈME D'EXPLOITATION

Critères de sélection

Origine de l'OS

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

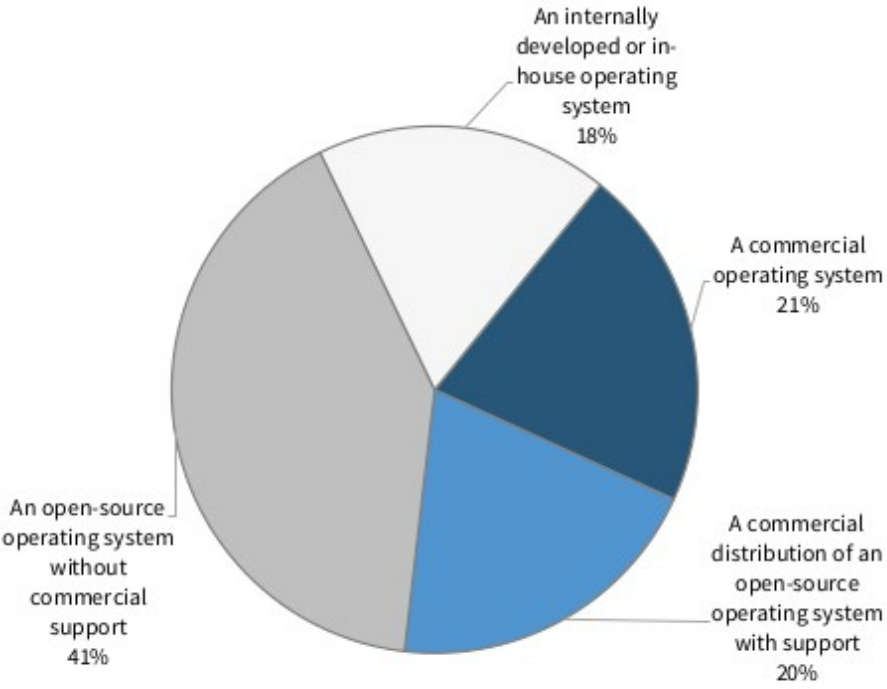
Critères de sélection

Application temps réel ?	→ RTOS → Extensions RT de Linux (LinuxRT, RTAI, …) parfois suffisantes
Mémoire limitée ?	→ OS léger
Ressources CPU limitées ?	→ RTOS à privilégier par rapport aux OS
Ressources énergétiques limitées ?	→ Si conso mémoire/CPU important, revoir les points précédents → un OS est plus enclin à posséder un outil de *power management*
MMU disponible ?	→ Oui = OS possible, de nombreux RTOS savent gérer une MMU → Non = prendre un « petit » OS, les RTOS gèrent généralement bien ça
Périphériques maisons ou de niche ?	→ Quel OS est compatible ? S'il y en a un … → Linux possède une compatibilité extrêmement large Ou avoir un développeur de driver Linux à portée de main
Sécurité ?	→ Vérifier la protection des tâches / *thread protected mode* / *process model*
Certification ?	→ Coût : généralement lié à la taille du code → Code source requis

Critères de sélection – étude 2023 eeTimes

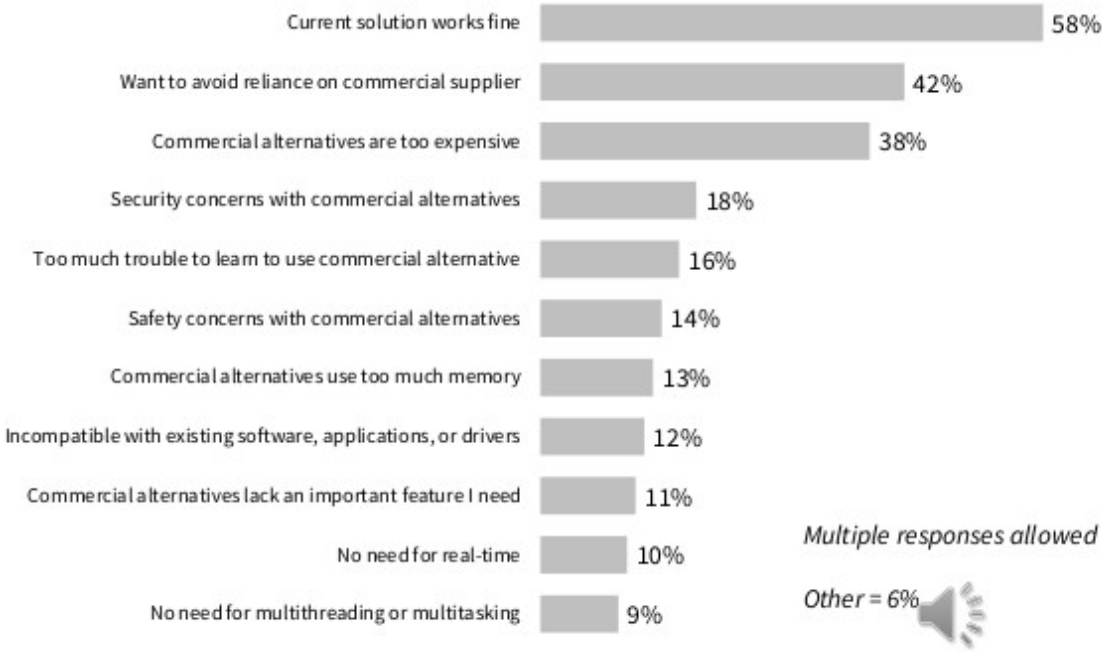
74% use an OS in current embedded project

OS Used in Current Embedded Project



Total Respondents

Reasons for not using commercial OS



Multiple responses allowed

Other = 6%

Base = Those not using commercial OS (284)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution propriétaire (commerciale)

Avantages

- Le coût implique *normalement* des contre-parties

→ maturité

→ support

→ engagement sur le long terme (portabilité, nouvelles architectures)

→ documentation (vérifier ! Code source dispo ? Lisible ?)

→ RTOS compatible avec outils de debug (outils de trace)

→ Drivers hardware

Inconvénients

- Coût de la licence

- Coûts cachés

→ Modules optionnels ? (payer le strict nécessaire)

→ Modules réglables ? Adaptables à la volée ?

- Pas de visibilité sur le fonctionnement interne

→ est-ce nécessaire ?

- Vérifier l'utilisation des standards (POSIX)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution libre

Avantages

- Coût de la licence
- Accès au code source

→ utile pour documentation

→ portabilité

→ peut aider à la certification

- Support de la communauté

→ Vérifier si communauté active

→ Vérifier sur quelle(s) version(s)

→ Vérifier sur quelle(s) architecture(s)

Inconvénients

- Problème de licence

→ restrictions des licences open source (GPL, LGPL, ...)

- Support à long terme pas forcément garanti

→ Juger selon l'activité de la communauté

- Trop d'implémentations

→ Une contribution = une nouvelle version

- Drivers moins fournis

→ Vérifier disponibilité middleware, BSP, ...

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution maison

Avantages

- Pas de coût de licence
- Maîtrise technique

→ impose de garder le savoir-faire

- Sur mesure

→ ou limité par les compétences de l'équipe …

Inconvénients

- Coût de développement
- Compétences en développement *kernel* nécessaires
- Support à long terme pas forcément garanti

→ Garder ses employés, faire la doc

- Portabilité

→ Compatibilité avec une nouvelle archi (asm, ISR, …) ?

- Drivers, middleware, ...

→ Augmente encore les coûts de développement

BONUS

ENSI CAEN

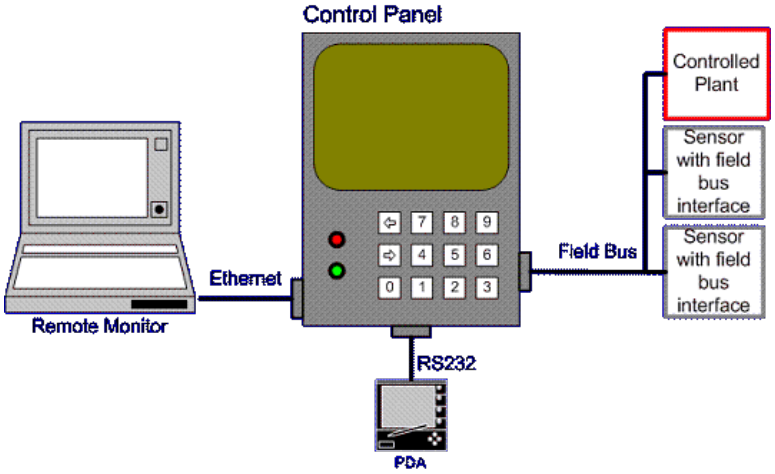
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Cas d’étude comparant 4 solutions pour répondre à un cahier des charges.

Disponible sur le site de FreeRTOS : https://www.freertos.org/tutorial/index.html

Applicable à n’importe quel RTOS.



Reference Manual - https://freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)
- Basile Dufay (MCF ESIX/UniCaen)