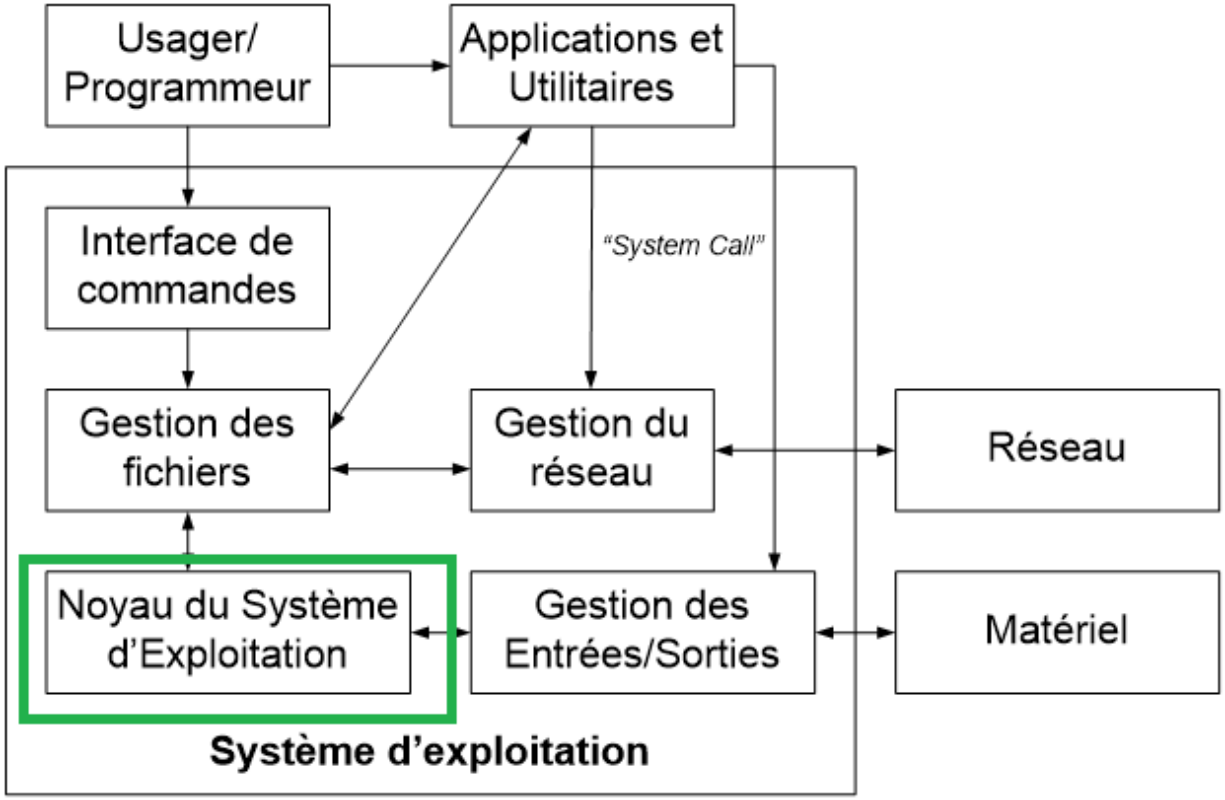
# Introduction

- Ce document explique comment est construit le scheduler d'un système d'exploitation préemption avec un cœur ARM Cortex M4:
  - Le rôle du système d'exploitation et le scheduler
  - Super boucle, tâches coopératives, OS non-préemptif et OS préemptif
  - Comment implémenter un OS préemptif
  - Implémentation dans le Cortex M4
  - Compétition des tâches pour les ressources: Mutex
  - Optimisation du temps de CPU avec OS préemptif: la fonction Wait en exemple
  - Accès aux périphériques avec OS préemptif
  - Le support du ARM Cortex M4 pour le OS
  - Les ressources requises par le noyau
  - Conclusion: Quand utiliser un OS préemptif, un OS non-préemptif, un scheduler ou des tâches coopératives

# Rôles du système d'exploitation



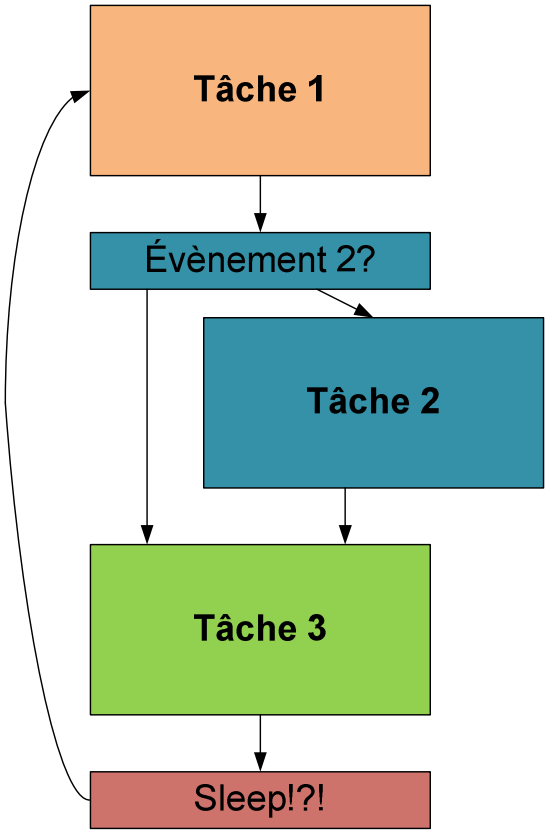

Parmi les nombreux rôles du système d'exploitation, ce document présente le noyau ou la partie responsable du temps de CPU

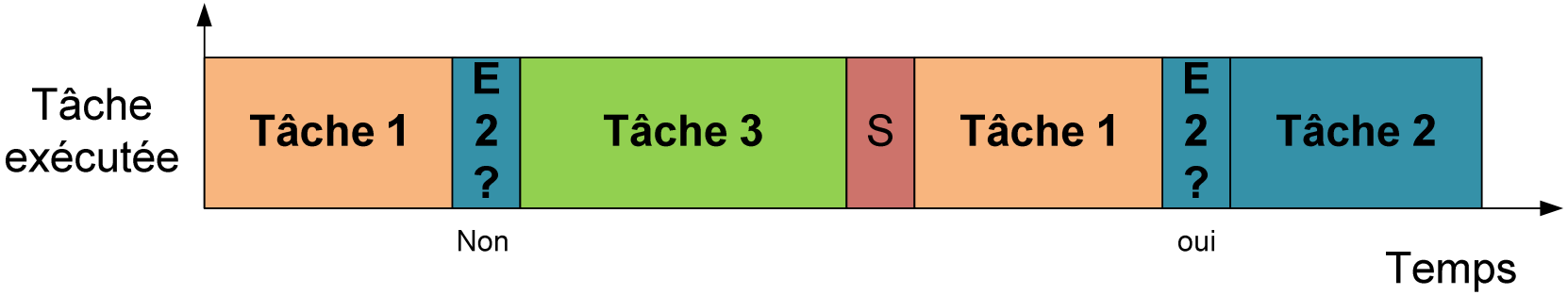
# Gestion du temps de CPU

- Une tâche attend souvent après un périphérique. Pendant qu'une tâche attend, il est possible d'exécuter d'autres tâches.
- Il y a souvent des délais dans les tâches. Pendant qu'une tâche attend, il est possible d'exécuter d'autres tâches.
- Une tâche attend parfois après une ressource utilisée par une autre tâche. Pendant qu'une tâche attend, il est possible d'exécuter d'autres tâches.
- Quand toutes les tâches sont finies ou en attente, il est possible d'économiser de l'énergie en mettant le processeur en veille.
- Certaines tâches peuvent être plus critiques ou plus prioritaires que d'autres…
- Il faut éviter d'exécuter des tâches inutilement.
- Plusieurs méthodes pour gérer le temps de CPU suivent!

# La super-boucle

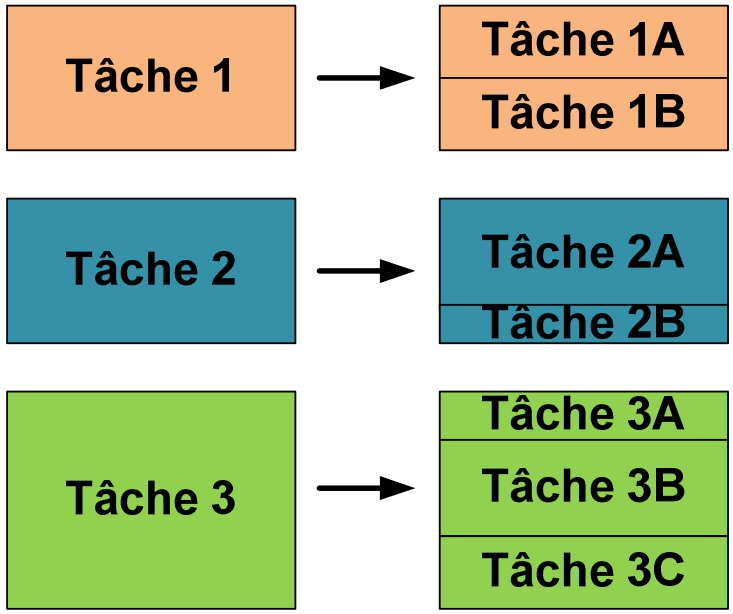
- La façon classique de gérer le temps de CPU est une super-boucle.
- Les tâches s'exécutent à tour de rôle dans un while(1)
- Des drapeaux sont levés lors d'interruptions des périphériques. Ils signalent des évènements traités dans la super-boucle.
- Il est possible de dormir un peu à la fin de chaque boucle pour sauver de l'énergie si les tâches sont courtes par rapport à la puissance de calcul du CPU.
- Le temps de microprocesseur est mal utilisé.

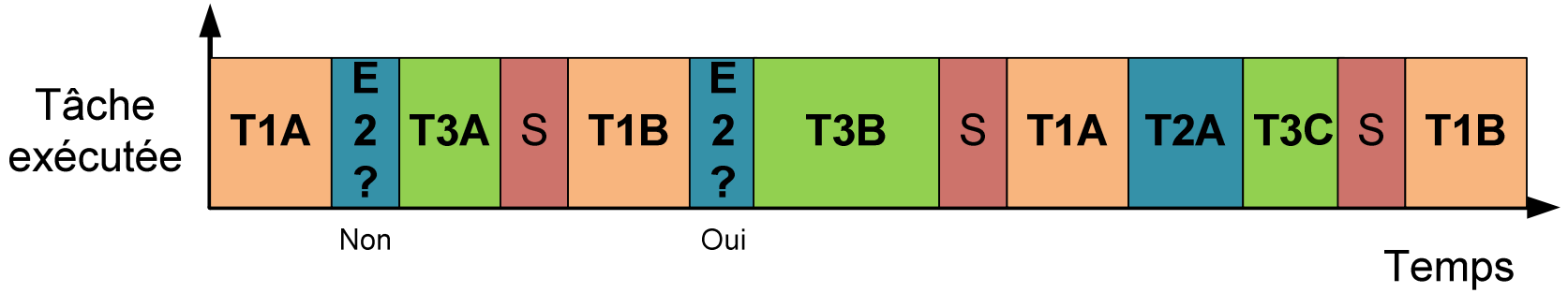


# Tâches coopératives

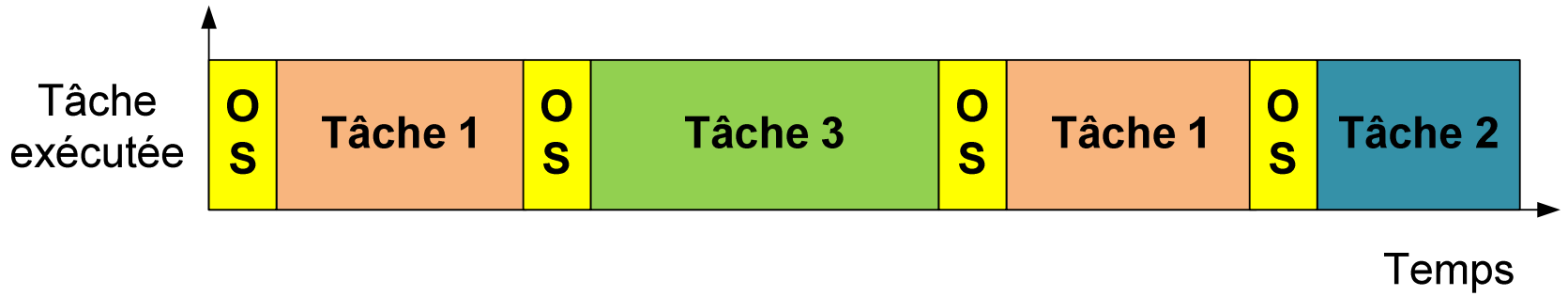
- Avec des tâches coopératives, les tâches sont découpées en parties inégales en utilisant des machines d'états.
- À chaque boucle de Main, on exécute une partie de chaque tâche.
- L'utilisation du temps de microprocesseur est encore non optimale



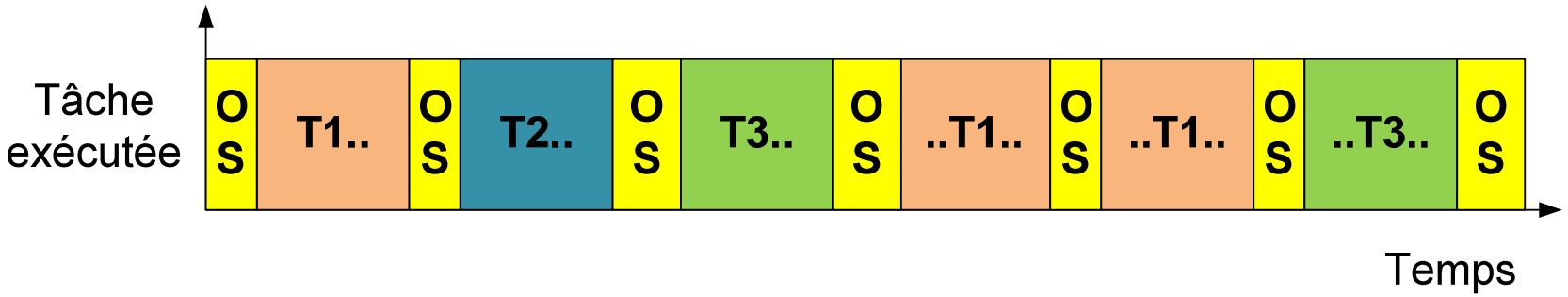
# Système d'exploitation non-préemptif

- Un système d'exploitation non préemptif décide la tâche à effectuer et donne le contrôle du microprocesseur à cette tâche.
- À la fin de chaque tâche, le programmeur (ou le compilateur) doit appeler une routine d'interruption (interruption logicielle) pour redonner le contrôle du système au système d'exploitation.
- Il est plus facile d'ajouter de nouvelles tâches dynamiquement avec un système d'exploitation non-préemptif qu'avec une super boucle ou des tâches coopératives.
- L'utilisation du microprocesseur n'est toujours pas optimale.



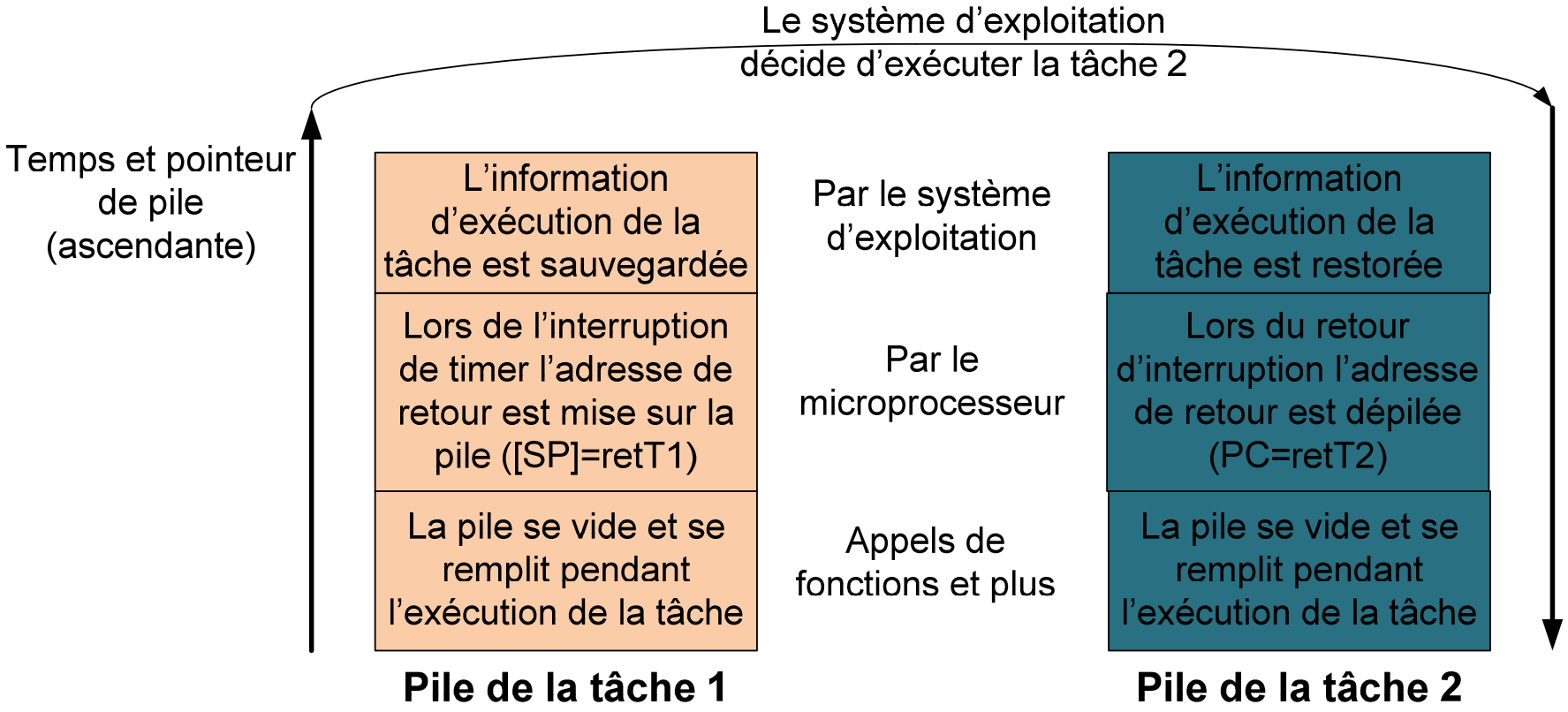
# Système d'exploitation préemptif

- Un système d'exploitation préemptif:
  - Interrompt la tâche en cours dans une interruption périodique de timer
  - Sauvegarde le contexte de la tâche interrompue pour en reprendre l'exécution ultérieurement
  - Décide la tâche à exécuter
  - Reprend l'exécution de la tâche à exécuter là où elle avait été interrompue
- Optimal:
  - Le système d'exploitation exécute seulement les tâches qui n'attendent pas (surtout après les périphériques) et qui ont les ressources pour être exécutées.
  - Il est possible de décider intelligemment de la tâche à exécuter pour respecter les contraintes du système!



# Comment ça fonctionne?

- Lors d'une interruption de timer l'adresse de retour de l'interruption est sauvegardée sur la pile. Il est possible de reprendre l'exécution d'une tâche là où elle a été interrompue. Pour cela, il faut aussi sauvegarder tous les registres et drapeaux du microprocesseurs pour la tâche…
- ***Le secret est l'utilisation d'une pile par tâche:***



## Implémentation avec l'ARM Cortex M4

- Le document ***SMI_C11_Changement de contexte avec coeur ARM Cortex M4*** sera présenté.

## Autres Implémentations

- Si désiré, le document ***C7a Atelier sur Fork et pThread.doc*** sera présenté.

# Décider de la prochaine tâche à exécuter

- Il y a plusieurs algorithmes possibles pour décider de la prochaine tâche à exécuter. Par exemple, on peut décider d'exécuter les tâches à tour de rôle ou par priorité. Les différents systèmes d'exploitations utilisent des stratégies différentes.
  - ***Dans tous les cas, les systèmes d'exploitation ne devraient jamais exécuter des tâches bloquées (par une autre tâche par exemple) ou en attente (d'un périphérique ou pour un délai).***
- Dans un RTOS, il y a des tâches critiques qui doivent être exécutées à l'intérieur d'un temps fixe. Les algorithmes d'ordonnancement des tâches diffèrent un peu entre les RTOS et les OS.
- Le système d'exploitation doit sauvegarder et mettre à jour des informations sur les tâches afin de décider de la prochaine tâche à exécuter:
  - État de chaque tâche
  - Dernière tâche exécutée
  - Priorité
  - …
- Pour économiser de l'énergie, le système d'exploitation peut choisir de mettre le microprocesseur en veille jusqu'à la prochaine interruption de timer du OS…

# C’est magique, mais…

- L’exécution de tâches concourantes peut entraîner des interactions indésirables entre les tâches:
  - Une interruption peut survenir n’importe quand
  - Une tâche peut interrompre une autre tâche n’importe quand.
    - Il faut éviter les variables globales
    - Il faut que les fonctions communes à deux tâches soient réentrantes
    - Il faut éviter d’écrire une variable dans deux contextes différents
    - …
- Les tâches peuvent compétitioner pour des ressources. Le système d’exploitation doit fournir des mécanismes pour gérer l’accès aux sections critiques. Le document ***SMI_C11_Notes sur les Mutex.doc*** sera présenté
- Une tâche mal codée peut encore gaspiller du temps de microprocesseur en faisant de l’attente active. Le document ***SMI_C11_Wait_Avec_OS_Preemptif.doc*** sera présenté.
- Les interruptions du système d’exploitation prennent du temps de CPU
- Il faut beaucoup de RAM pour les piles des tâches

# Gestion des interruptions et des périphériques

- Avec un système d'exploitation préemptif, toutes les interruptions du système (et tous les transferts par DMA) sont gérées par le système d'exploitation.
- Toutes les interruptions des périphériques (et les autres aussi!) ont une priorité plus haute que celle du timer du système d'exploitation
  - Pourquoi?
- Pour accéder aux périphériques, les tâches doivent passer par une fonction du système d'exploitation qui:
  - Vérifie si le périphérique est disponible et réserve le périphérique s'il l'est (Mutex)
  - Initie l'accès au périphérique
  - Change l'état de la tâche pour éviter que la tâche ne soit exécutée inutilement pendant qu'elle attend après le périphérique
  - Attend (le système d'exploitation exécute alors d'autres tâches)
  - Attend (une interruption du périphérique survient)
  - Attend (le système d'exploitation gère le périphérique et change l'état de la tâche)
  - Retourne le résultat de l'accès au périphérique à la tâche appelante

# Admission de tâches

- Dans les exemples présentés, les tâches sont ajoutées au système de manière statique (par le compilateur). Or, tous les systèmes d'exploitation permettent d'ajouter des tâches dynamiquement. Pour ajouter une tâche, le système d'exploitation doit:
  - Réserver de la mémoire pour la tâche (code, données et pile)
  - Réserver de la mémoire pour de l'information sur la tâche (état, priorité, emplacement dans la mémoire, pointeur de pile et plus!)
  - Mettre la tâche ou une partie de la tâche en mémoire pour l'exécuter
  - Initialiser la pile de la tâche pour « retourner » au début de le tâche quand le système d'exploitation décidera de l'exécuter
  - Mettre l'état de la tâche à « Prêt » pour décider, éventuellement, d'exécuter la tâche lors de l'interruption de timer du OS.

# Support au Système d'Exploitation (bis)

- Le Cortex M4 a plusieurs caractéristiques utiles pour construire un petit système d'exploitation. Ces caractéristiques permettent d'implémenter un RTOS facilement.
  - **Systick Timer**: Un timer périodique pour générer les interruptions du OS dans lesquels on détermine la prochaine tâche.
  - **Mode handler/thread**: Lors d'une interruption, le microprocesseur entre en mode « handler », c'est-à-dire en mode OS (qui doit gére habituellement toutes les interruptions…). Sinon, le microprocesseur est en mode thread. Il exécute les instructions du main en arrière-plan.
  - **Banque de registres SP**: Permet de sauver de la mémoire RAM avec un système d'exploitation préemptif (voir ***SMI_C11_Changement de contexte avec coeur ARM Cortex M4***) .
  - **Memory Protection Unit**: L'unité de protection de la mémoire permet de rendre certaines zones de la mémoire accessible unique par le OS ou par un programme spécifique.
  - **Temps de latence pour entrer dans les interruptions réduit au minimum**: Réduit le coût, en terme de % de temps de CPU, du OS préemptif.
  - **Sauvegarde de contexte ultra rapide**: LDM, STM

# Les ressources utilisées par un OS préemptif

- Le code du scheduler est souvent très petit (la sauvegarde de contexte se fait en quelques instructions par exemple). Il faut habituellement 1KB à 2KB de mémoire code pour le scheduler
- Par contre, il faut une pile par tâche et une pile pour les interruptions… La grosseur de la pile requise pour chaque tâche dépend de la tâche, mais elle doit souvent être surdimensionnée pour palier à toutes les éventualités. Un OS avec quatre tâches et des piles de 2KB aura possiblement de 11KB de RAM (4*2KB + 3KB pour les informations sur les tâches et la pile des intrerruptions.
- La plupart des petits systèmes d'exploitation requièrent 10KB à 12KB de code, surtout pour les pilotes de périphériques, le système de fichier et les autres parties du système d'exploitation.

# Conclusion

- De plus en plus de systèmes embarqués utilisent un système d'exploitation ou une forme de scheduler (environ 70%): le coût pour supporter un OS est de plus en plus petit par rapport aux ressources des microcontrôleurs.
- Il y a plusieurs raisons qui motivent l'utilisation d'un système d'exploitation
  - Meilleure utilisation du temps de CPU
  - Beaucoup de fonctions déjà codées (drivers, système de fichier, réseau, affichage…)
  - Permet d'exécuter de nouvelles tâches dynamiquement (possible aussi avec la superboucle, mais compliqué!)
  - Clarté du code (pas de machine d'état comme dans les tâches coops)
  - …
- Malgré cela, il faudra toujours planifier les séquences de tâches adéquatement et leurs priorités: un système d'exploitation permet de mettre des priorités sur les tâches, mais il revient quand même au programmeur de faire en sorte que son système respecte les contraintes de temps et d'exécution de l'application désirée.