CEG4566/CSI4541 – Conception de systèmes temps réel

Chapitre 9 – Exécution concurrente

Références principales:
Real-time systems and programming languages, Burns and Wellings
Fundamentals of Embedded Software: Where C and Assembly Meet, Prentice-Hall, 2001, Daniel W. Lewis

9.1 Programmation concurrente et programmation temps réel

9.1.1 Rappel : Programmation concurrente

On appelle programmation concurrente les techniques et notations permettant :

- L'expression et la manipulation (construction, destruction, lancement, arrêt, ...) d'entité concurrentes que l'on appelle tâches ou processus.
- La mise en place de moyens de communication et de synchronisation entre ces entités.
- L'abstraction de l'exécution réelle des processus ou tâches sur une architecture parallèle ou non

9.1.2 Rappel : Programmation temps réel

La programmation temps réel est assimilable à la programmation concurrente avec comme particularité :

- Des mécanismes ou notations permettant d'accéder au temps réel (généralement une horloge temps réel).
- Des mécanismes permettant de définir ou de modifier la priorité des processus et/ou de définir et de modifier des échéances temporelles.
- La suppression ou l'adaptation de certains traits du langage utilisé afin d'alléger l'applicatif et/ou de le rendre plus déterministe.

9.1.3 Intérêts de la programmation Concurrente

- Optimisation de l'utilisation des ressources

- La traduction d'un monde à base d'entité concurrentes à l'aide de techniques de programmation séquentielle.

9.1.4 Problèmes inhérents à la programmation concurrente

➔ ***Problèmes de sûreté*** : La synchronisation des processus ou tâches peut entraîner des problèmes tels que l'interblocage ou la famine

- *Par exemple, le processus A attend le processus B, qui lui même attend le processus C qui lui même attend le processus A : il y a un interblocage (ou étreinte fatale ou encore deadlock)*
- *Autre exemple, le processus A et B forment une coalition pour empêcher C d'accéder à une ressource (A et B se la passent) : il y a famine du processus C*

➔ ***Problèmes inhérents à la concurrence***

- *Problèmes de vivacité* : Suite à une mauvaise entente, il est possible que deux processus n'arrivent jamais à se synchroniser; ceci peut conduire à l'absence de respect d'un objectif fixé sans pour autant qu'il y ait interblocage. Les deux processus sont actifs (ils viennent régulièrement) mais ils ne font pas progresser les choses : on est face à une activité non constructive et donc un problème de vivacité.
- *Difficulté à reproduire les erreurs* : L'entrelacement des différents processus (chacun peut avancer à son rythme) va définir un nombre de comportements possibles très important.
  - *Si l'on a quatre processus indépendant, chacun ayant dix états différents, on obtient un système complet ayant 10^4 = 10 000 états différents*
  - *Chaque exécution ne va pas nécessairement faire évoluer exactement de la même façon les processus (leur évolution dépend généralement du monde extérieur qui est en perpétuel changement)*

  Ainsi, si une erreur survient il est très difficile de reproduire cette erreur afin de comprendre son origine

9.1.5 Côtés positifs de la programmation concurrente

- La concurrence introduit des problèmes spécifiques qui peuvent être difficile à analyser

- Cependant, elle simplifie énormément le travail du concepteur et du programmeur qui peut à la fois calquer la réalité (traitement parallèle, simultané) dans son programme tout en tirant partie de l'optimisation possible des ressources.

- Il existe de plus des méthodes de test formel adaptées à l'analyse de systèmes concurrents

9.1.6 Notes

- La programmation concurrente simplifie la conception des systèmes temps-réel qui sont par nature des systèmes concurrents.

- Elle peut cependant introduire certains problèmes spécifiques liés en grande partie à la synchronisation des tâches.

- Les langages, ou les systèmes d'exploitation, offrent différents mécanismes pour la représentation et l'exécution des tâches ou processus.

9.2 Concurrence *vs* parallélisme :

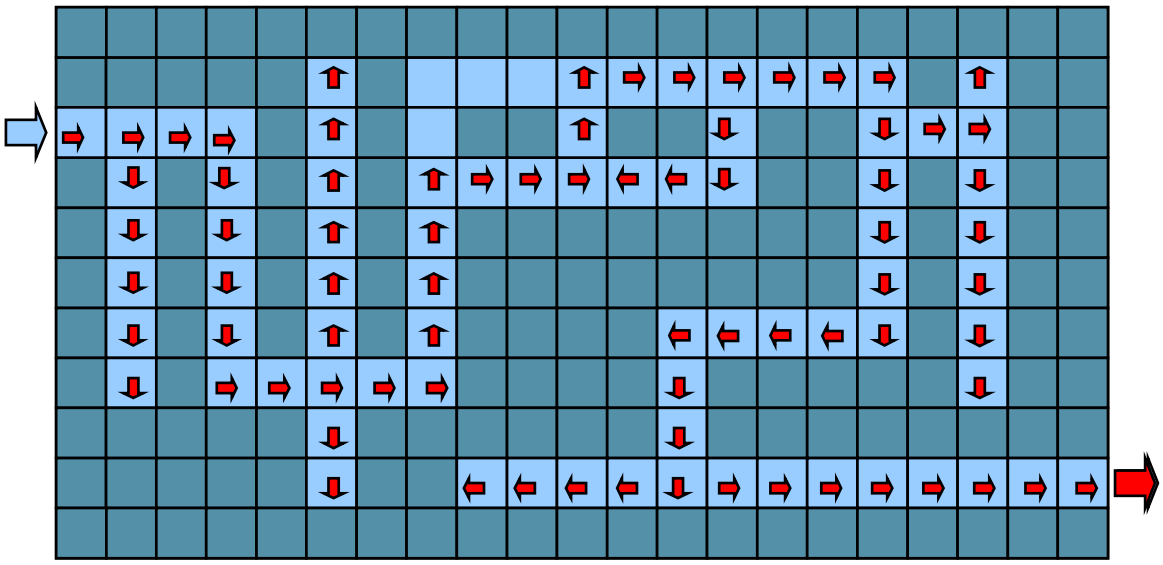
- Concurrence : Traitement logique simultané, ne nécessite pas des unités de traitement multiples.
- Parallélisme : Traitement physique simultané, nécessite des unités de traitement multiples et indépendantes.

Note : Dans les deux cas, on a besoin de contrôler l'accès aux ressources partagées.

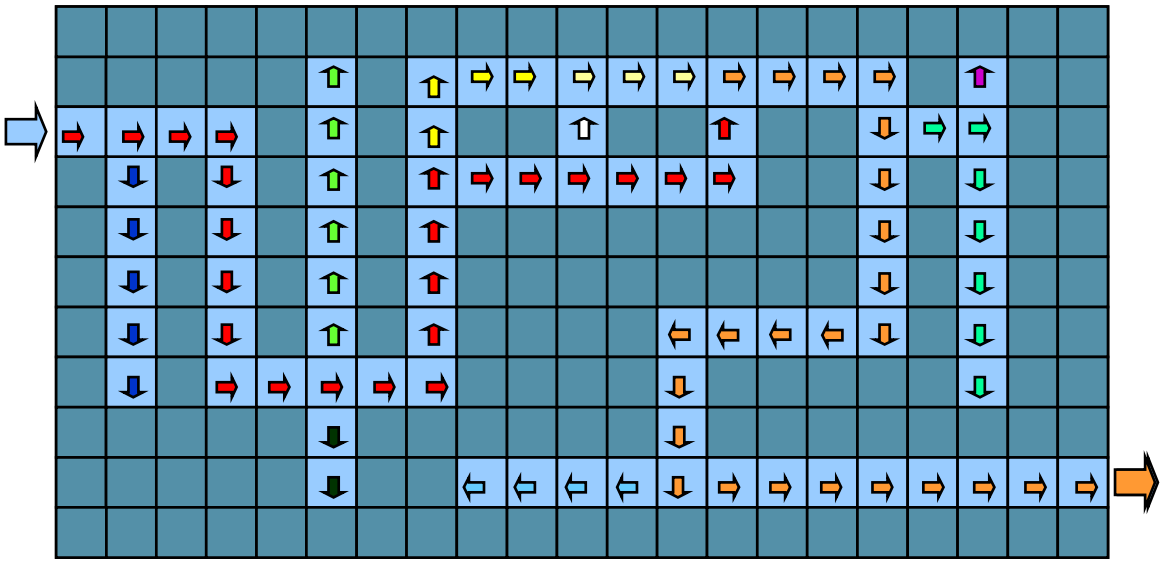
9.3 Concurrent *vs* séquentiel :

Exemple trouver son chemin dans un labyrinthe :

Recherche séquentielle



Recherche parallèle



9.4 Construction de la programmation concurrente

Même si les concepts de programmation concurrente différent d'un langage à un autre et d'un OS à un autre, il reste qu'on doit assurer les fonctions suivantes :

- La définition de l'exécution concurrente se fait à travers la notion de processus.
- Synchronisation de processus.
- Communication interprocessus.

En considérant les interactions entre processus, on distingue 3 comportements différents :

- L'indépendance
- La cooperation
- La competition

9.5 Exécution concurrente

Même si la notion de processus telle qu'énoncée dans ce cours est commune à tous les langages concurrents, il y a des différences dans les modèles de concurrence adoptés. Ces différences résident dans :

9.5.1 La structure :

Statique (nombre de processus prédéfini), dynamique (processus créés au besoin)

9.5.2 Le niveau :

Deux cas pour le niveau de parallélisme supporté:

- Les processus peuvent être définis à n'importe quel niveau du programme et même dans d'autres processus (*nested*) ou multi-niveau. Notions de *parent/enfant* et de *gardien/dépendant (ou maitre/esclave)*.

- Les processus sont définis au niveau haut du programme uniquement (*flat*)

Quelques exemples :

Langage	Structure	Niveau
Pascal concurrent	Statique	*flat*
Occam2	Statique	*nested*
Modula-1	Dynamique	*flat*
Modula-2	Dynamique	*flat*
Ada	Dynamique	*nested*
Java	Dynamique	*nested*
Mesa	Dynamique	*nested*
C/POSIX	Dynamique	*flat*

9.5.3 La granularité :

- Grossière (Ada, POSIX processes/threads, Java): contient relativement peu de processus
- Fine (occam2): contient plus de processus.

9.5.4 L'initialisation :

Passement de paramètres, Communication interprocessus IPC

9.5.5 La terminaison :

Elle se fait par :

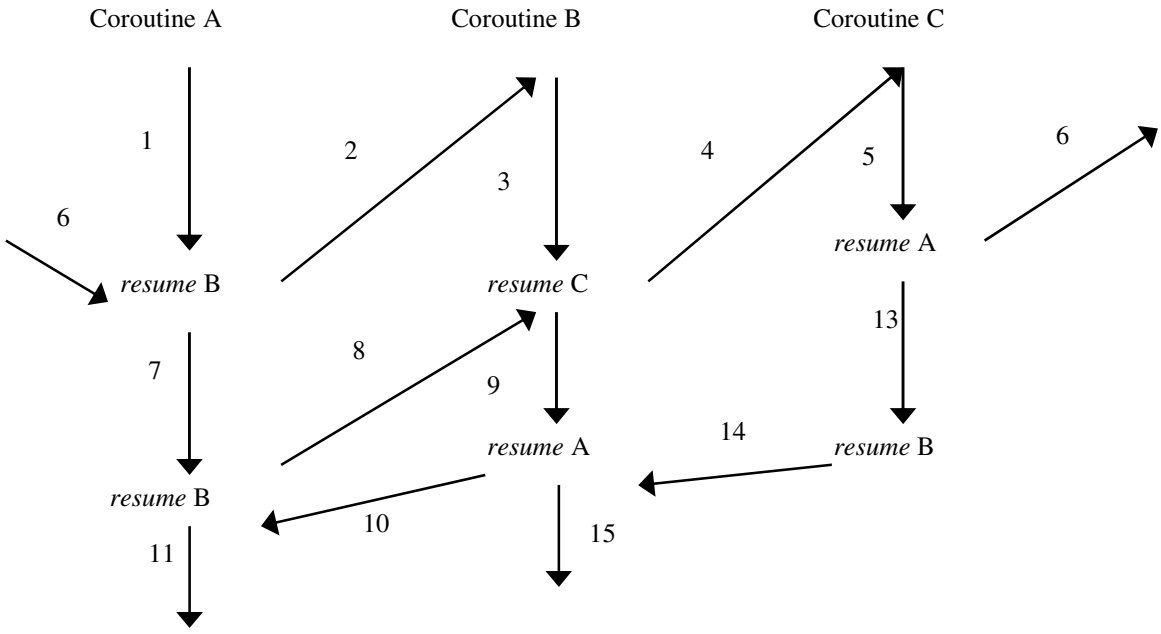
- Fin du de l'exécution du processus;
- Suicide par l'exécution d'une auto-terminaison;
- Avortement à travers une action explicite provenant d'un autre processus;
- Apparition d'une erreur d'exécution;
- Processus dans une boucle sans fin;
- Plus besoin de lui.

9.5.6 La représentation :

9.5.6.1 Les Coroutines

Elles sont comme des sous-programmes, il n'y a pas de hiérarchie. Pas de '*return*' mais plutôt une déclaration '*resume*' qui veut dire *reprendre* la coroutine indiquée.

Question : Est-ce que les coroutines implémentent un vrai parallélisme?



9.5.6.2 *Fork* et *Join*

- *Fork* veut dire qu'une routine spécifiée doit commencer à s'exécuter concurremment avec celle qui l'a invoqué.

- *Join* permet à une routine d'attendre la fin de l'exécution de la routine qu'elle a invoquée.

```
function F return is ...;
procedure P;
      ...
      C:= fork F;
      ..
      J:= join C;
      ...
end P;
```

Après le *fork*, P et F vont s'exécuter concurremment. Au niveau de *join*, P va attendre jusqu'à ce que F ait fini (si ce n'est pas déjà fait).

Les notations *Fork* et *join* peuvent être trouvées dans Mesa et UNIX/POSIX

9.5.6.3 Cobegin

Cobegin (ou *parbegin* ou *par*) est une façon structurée de montrer l'exécution concurrente entre une série de déclarations.

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  Sn
coend
```

- S1, S2 etc., s'exécutent concurremment
- La déclaration se termine quand S1, S2, etc. sont terminées
- Chacun des Si peut être n'importe quelle déclaration acceptée par le langage de programmation.
- Cobegin peut être trouvée dans Edison et occam2.

9.5.6.4 Déclaration explicite du processus

- La structure d'un programme peut être plus claire si on sait si les routines vont être exécutées concurremment.

- À noter que ceci ne dit pas quand elles vont être exécutées.

```
task body Process is
begin
...
end;
```

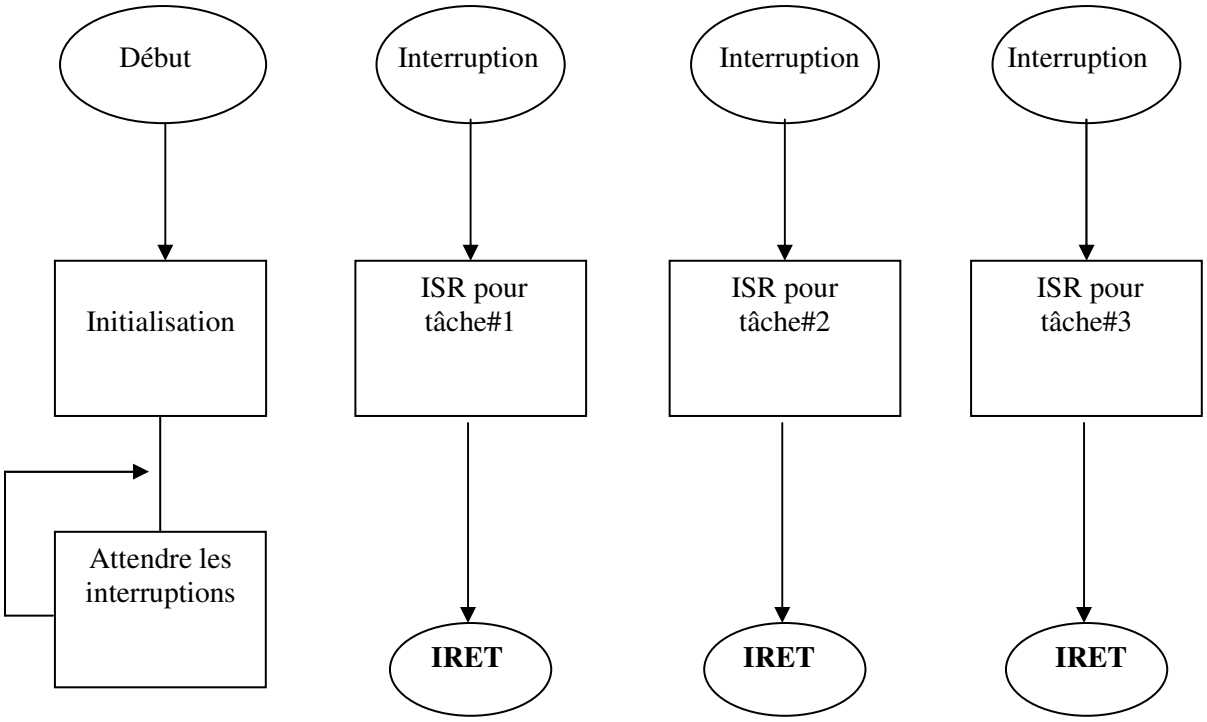
Les langages qui incluent la déclaration explicite de processus peuvent avoir des créations de processus/taches implicites ou explicites.

9.6 Les langages concurrents

- Ce sont des langages généralistes incluant de plus la notion de tâches et des primitives de synchronisation
- Ils ont un haut pouvoir d'abstraction, indépendance des architectures et des systèmes cibles (ou très peu dépendant)
- Parmi ces langages, Ada est sans doute le langage le plus abouti mais des restrictions de Java peuvent être utilisées à profit dans le développement d'applications/systèmes temps réel.
- ADA et Java sont des langages à privilégier lorsque d'autres contraintes (manque de formation, reprise de code existant, coopération inter-équipes/ ou inter entreprises, ...) ne rendent pas la chose difficile.
- Tout langage préemptif et gérant les exceptions (interruptions) peut être utilisé.

9.7 Organisation d'un système en *tâche premier plan* et en *tâche de fond* (*Foreground/Background*)

- Le gros du travail est fait en mode *foreground* par les ISRs (*Routine de Service d'Interruption*). Chaque ISR s'occupe d'un événement matériel particulier (cas des systèmes embarqués).
- Le programme principal après initialisation entre dans une boucle en tâche de fond (*background*) et attend que les interruptions apparaissent.
- Permet au système de répondre à des événements externes avec une certaine latence prédictible (à discuter la notion de temps pire d'exécution, *WCET: Worst Case Execution Time*).



Transférer le travail vers une tâche de fond

- Déplacer un travail non critique vers une tâche de fond (mode background), par exemple un affichage d'information non critique sur un écran.
- L'ISR en *foreground* écrit la donnée dans une file d'attente, la tâche en *background* la prend et la traite.
- Ceci est une alternative au fait d'ignorer une interruption pour cause de trop de sollicitations aux entrées (trop de demande d'interruption, certaines demandes seront ignorées).

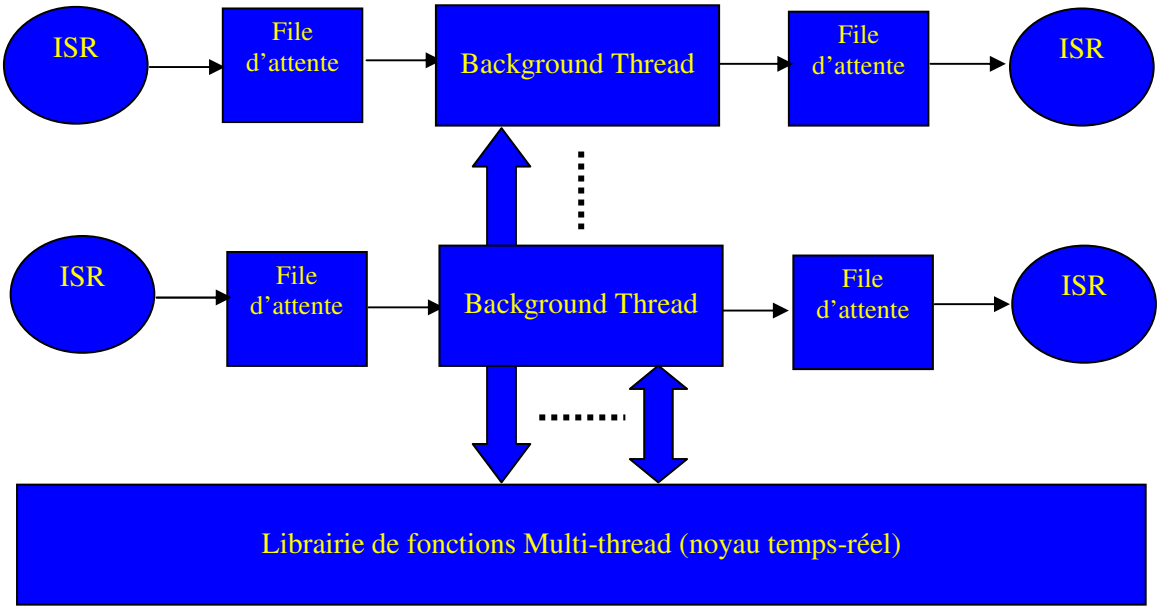
Limitations et inconvénients de transférer le travail vers une tâche de fond

- Pour assurer la meilleure performance possible, il faut transférer le maximum de travail vers les tâches de fond.
- Le background devient une collection de files d'attente et de routines qui traitent les données.
- Optimise la latence de chaque ISR mais les tâches background doivent gérer et partager l'accès au processeur.

Exemple à discuter: Affichage d'information sur le Garmin G1000 PFD (*Primary flight Display*)

9.8 Architecture Multi-Thread

9.8.1 L'architecture

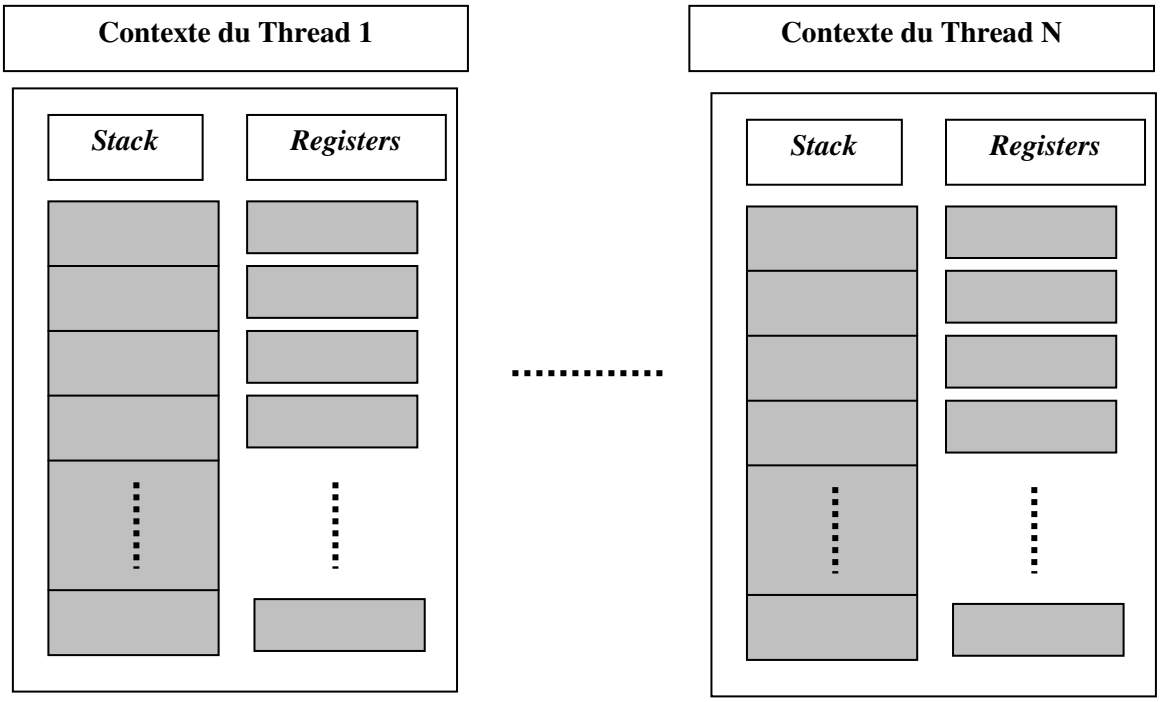


9.8.2 Conception des threads

- En générale, le thread passe par une initialisation puis entre dans une boucle infinie de traitement.
- Au début de la boucle le thread renonce au processeur en attendant l'arrivée de données, l'apparition d'un événement externe ou une condition qui devient 'vraie'

9.8.3 Exécution concurrente de threads indépendants

- Chaque thread s'exécute comme s'il avait sa propre CPU, séparé des autres threads.
- Les threads sont conçus, programmés, et se comportent comme s'ils étaient le seul thread à s'exécuter.
- Le partitionnement du background en un ensemble de threads indépendants simplifie chacun des threads et par conséquent la complexité du programme.
- Chaque thread maintient sa propre pile et le contenu de ces registres.



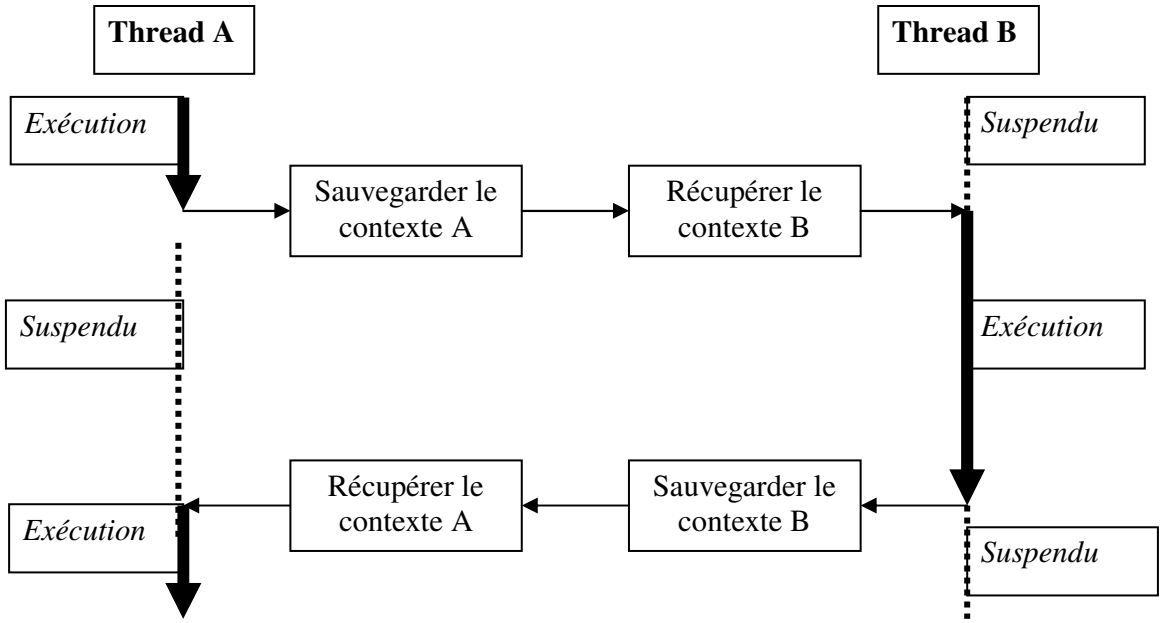
9.8.4 La concurrence et la simultanéité

- Seul un thread est exécuté, les autres sont suspendus.
- Le processeur passé d'un thread à un autre rapidement ce qui donne l'impression que les threads s'exécutent simultanément. On dit qu'ils s'exécutent concurremment.
- Le programmeur assigne les priorités à chacun des threads et l'ordonnanceur utilise cette priorité pour déterminer quel thread va être le prochain à s'exécuter.

9.8.5 Comment ça fonctionne?

- Les threads appellent une librairie de routines *run-time* du noyau temps-réel.
- Le noyau fournit les mécanismes de passage d'un thread à un autre, la synchronisation, la communication et la priorité.

9.8.6 Changement de contexte



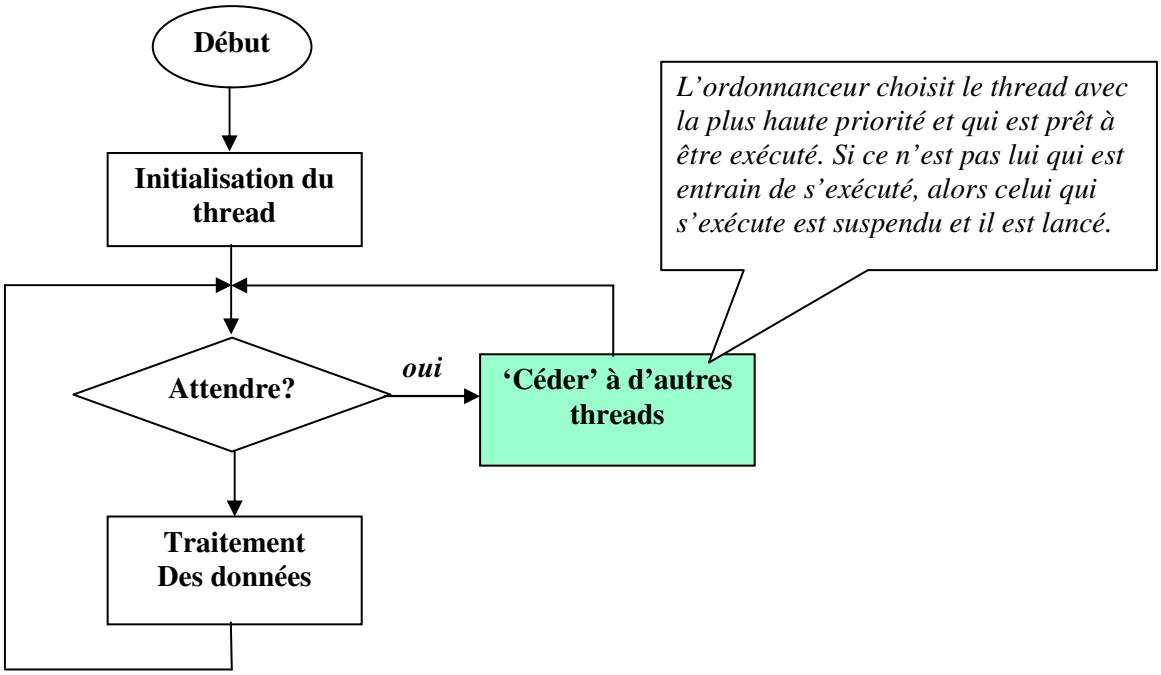
9.8.7 Multitâche non préemptif

Comment gérer le multitâche dans un système non préemptif?

- Les threads appellent une routine du noyau pour réaliser un changement de contexte.
- Le thread renonce au processeur, ce qui laisse un autre thread l'utiliser et s'exécuter.
- Le changement de contexte est base sur le principe de '*céder*" et ce multitâche est appelé '*multitâche coopératif*'.
- Quand un événement externe apparait, il se peut que le thread qui est entrain de s'exécuter ne soit pas celui qui a été conçu pour gérer cet événement.
- La première possibilité pour que le thread approprié s'exécute et après que le thread en exécution arrive au '*céder*" suivant (on rappelle qu'on est en situation non préemptive).
- Quand 'céder' arrive, il se peut que d'autres threads soient programmés pour s'exécuter en premier.
- Dans la majorité des cas, ceci rend impossible de prédire le temps maximum de réponse d'un système multitâche non préemptif.

- Le programmeur doit prendre ça en compte et appeler la routine 'céder' fréquemment, sinon le temps de réponse du système pourrait poser des problèmes.
- 'céder' doit être insérer dans toute boucle où un thread est en attente d'une condition/événement externe.
- On aura besoin aussi de 'céder' dans une boucle qui prend beaucoup de temps (exemple: lire ou écrire dans un fichier), ou bien distribué périodiquement dans les traitements très long.

Changement de contexte dans un système non préemptif :



9.8.8 Multitâche préemptif

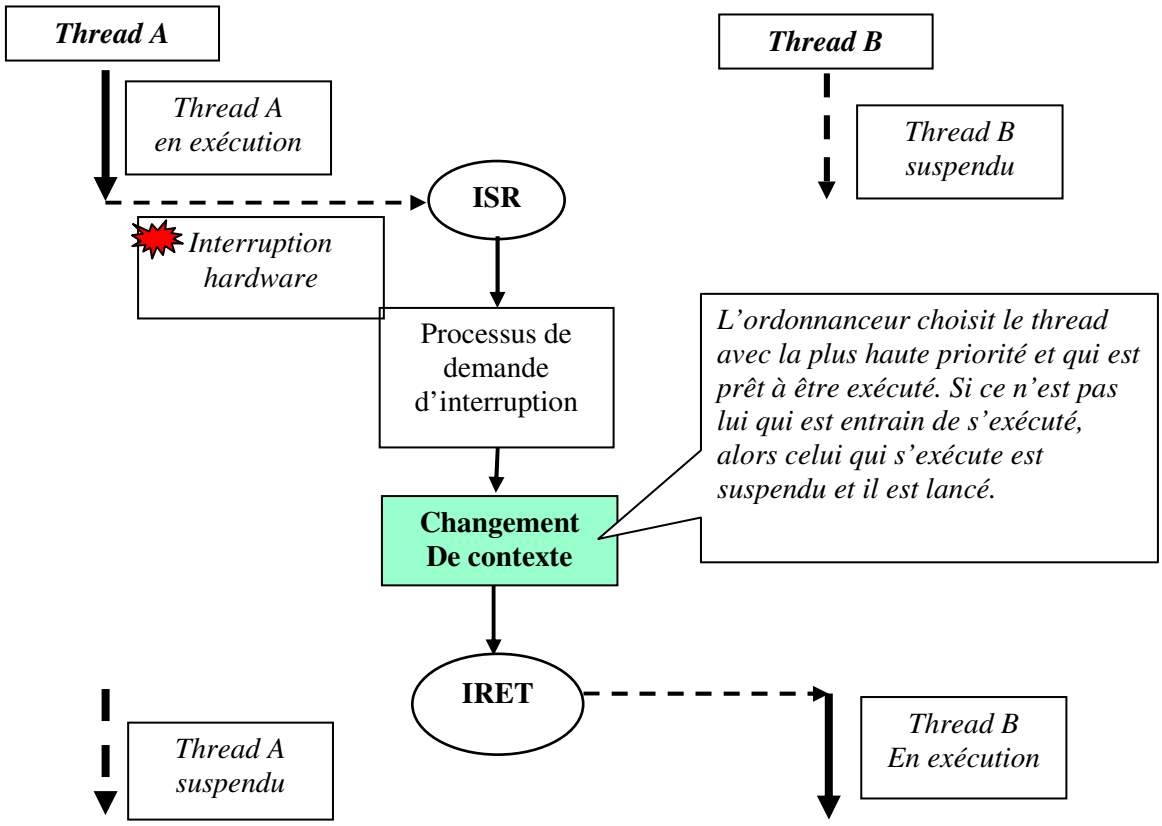
- Les interruptions hardware lancent le changement de contexte.
- Quand un événement externe apparait, une ISR hardware est invoquée.
- Après avoir servi la demande d'interruption l'ISR change le contexte vers le thread de plus haute priorité et qui est prêt à s'exécuter et y retourne.

Avantages :

- Améliore considérablement le temps de réponse du système.
- Élimine l'obligation pour le programmeur d'inclure des appels explicites au noyau pour gérer le changement de contexte entre les différents threads en background.

- Le programmeur n'a plus besoin de prévoir le changement de contexte. Le changement de contexte se fera quand c'est nécessaire uniquement en réponse à des événements externes.

Changement de contexte dans un système préemptif :



9.8.9 Threads, ISRs, et partage

1. **Entre un thread et une ISR:**

Une dégradation des données peut avoir lieu si la section critique d'un thread est interrompue pour exécuter une ISR.

2. **Entre 2 ISRs:**

Une dégradation des données peut avoir lieu si la section critique d'une ISR est interrompue pour exécuter une autre ISR.

3. **Entre 2 threads:**

Une dégradation des données peut avoir lieu sauf si leurs sections critiques sont correctement coordonnées.