Informatique industrielle A7-19571
Systèmes temps-réel

J.F.Peyre

Partie II : Programmation concurrente

Plan

- Définitions
- Intérêts de la programmation concurrente
- Problèmes possibles
- Généralités sur la manipulation de processus et de tâches
- Ecriture et lancement de processus et de tâches
- Conclusion

Quelques définitions

Programmation concurrente : définition

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

Programmation temps réel : définition

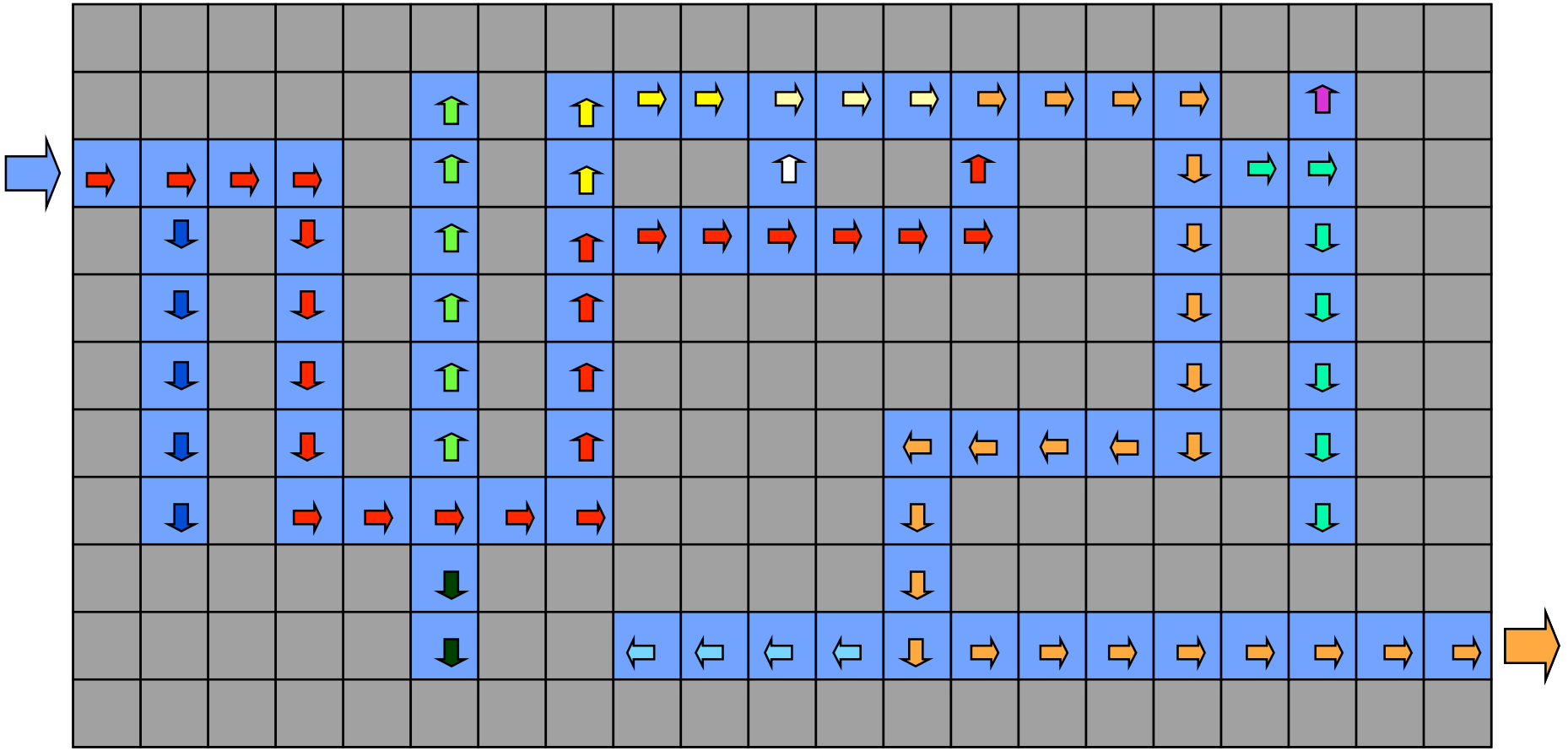
La programmation temps réel est assimilable à la programmation concurrente avec comme particularité :

- 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

Intérêts de la programmation concurrente

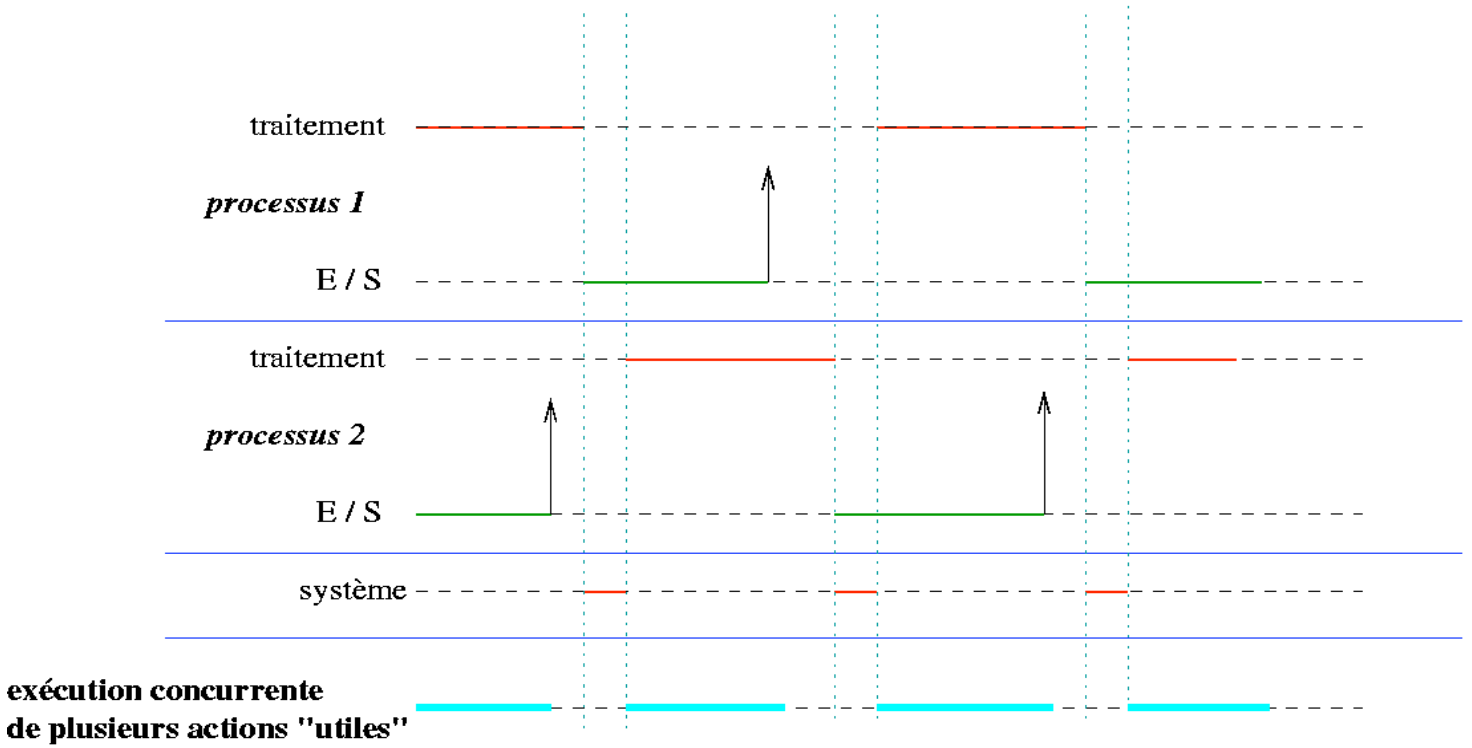
Optimisation de l'utilisation des ressources

Parallélisation d’une recherche



Optimisation de l'utilisation des ressources (suite)

Optimise l'utilisation du CPU sur une architecture mono-processeur par la prise en compte du parallélisme entre calcul et entrées / sorties



Calquer le monde réel

Le monde réel est fait d'entité concurrentes qui interagissent

- Un système de réservation de billets doit nécessairement prendre en compte le fait que les agences effectuent les réservations en parallèle.

- Un système de guidage doit utiliser les informations fournis par plusieurs capteurs simultanément et donner des ordres à plusieurs actionneurs

Calquer le monde réel (suite)

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

- inclure dans le code la répartition de l'appel des différents composants du système
- modifier le programme dès que l'on change d'architecture (nombre de processeur, répartition de l'application)
- modifier profondément le programme dès que l'on inclu une nouvelle activité

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*
- **Ces problèmes sont 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**

  Par exemple, pour réparer un chauffe-eau électrique, le plombier peut exiger que l'électricité soit au norme : il faut donc l'intervention préalable de l'électricien; l'électricien peut lui exiger de ne faire les travaux que lorsque les problèmes de fuites d'eau seront résolus : il faut au préalable l'intervention du plombier

- **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**

Difficile reproductibilité des 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**

Utilisation de la concurrence : bilan positif

- 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 dans son programme tout en tirant partie de l'optimisation possible des ressources
- Il existe de plus des méthodes de test formel adaptés à l'analyse de systèmes concurrents

Généralités sur la manipulation de processus et de tâches

Terminologie

On dit qu'un système est :

- **Multi-tâches** lorsque plusieurs entités concurrentes s'exécutent sur une machine mono processeur
- **Parallèle** lorsque plusieurs entités concurrentes s'exécutent sur plusieurs processeurs avec une mémoire commune
- **Réparti** lorsque plusieurs entités concurrentes s'exécutent sur des machines distinctes reliées en réseau et ne partageant pas de mémoire commune

Terminologie (suite)

- **On dit que deux processus ou tâches sont**
  - indépendants si l'évolution de l'un n'influe pas et ne dépend pas de l'autre
  - en coopération si chacun mène un travail distinct de l'autre mais avec un objectif commun et la mise en place de synchronisation et de communication et qu'il n'y a pas compétition sur l'accès à certaine ressources
  - en compétition lorsqu'ils sont en coopération mais lorsque, de plus, ils sont en compétition sur l'accès de certaines ressources
- **Selon le niveau d'observation, deux processus peuvent être vus en coopération ou en compétition**

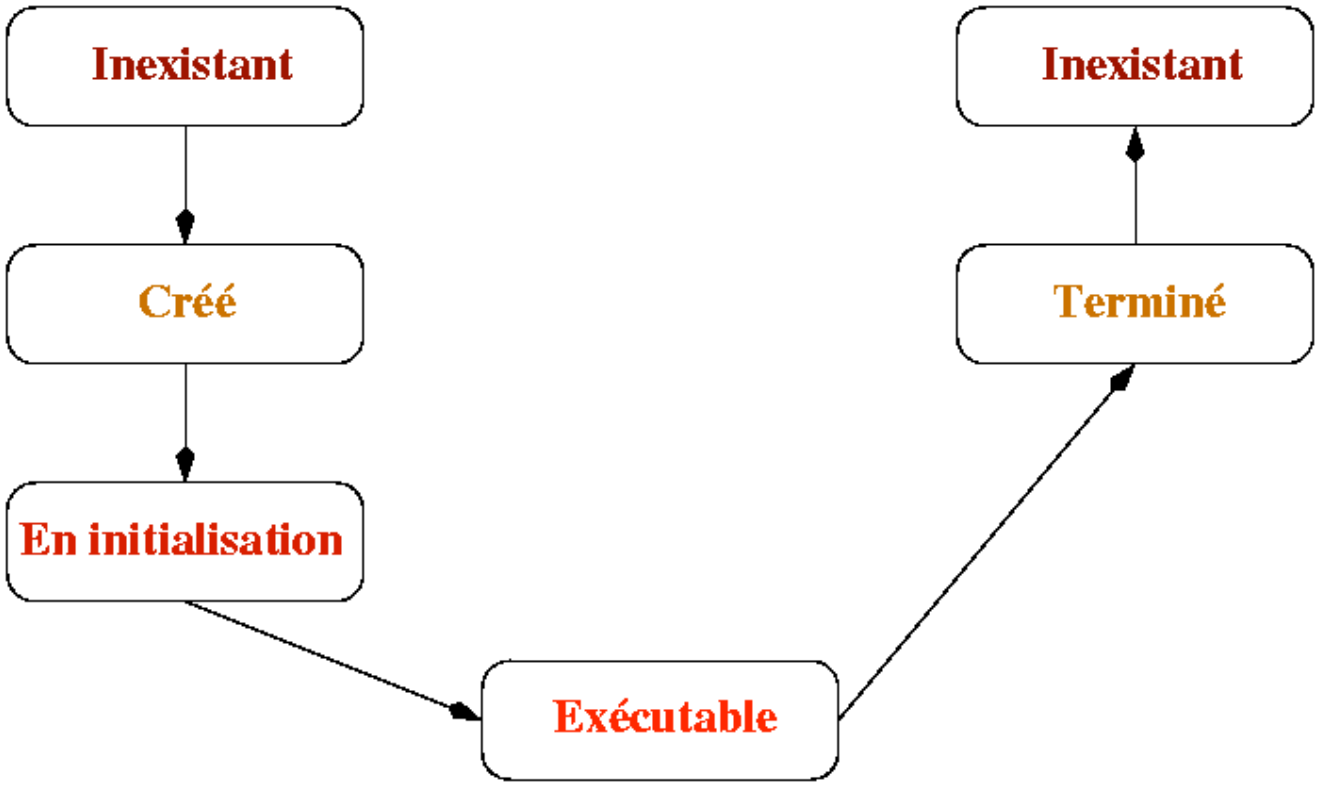
Processus *vs* tâches

- Un processus possède son propre espace d'adressage : il s'exécute sur sa propre machine virtuelle;
- Le système d'exploitation (ou le "run-time" dans le cas d'un exécutif) met en place des mécanismes de protection contre l'interférence inter-processus
- Le partage de mémoire est alors soit impossible soit doit se faire de manière explicite (i.e. appel système)

Processus *vs* tâches (suite)

- Une tâche partage son espace d'adressage (sa machine virtuelle) avec d'autres tâches (dépend du système et de la création des tâches)
- Les mécanismes de protection limitant l'interférence inter-processus n'existent pas pour les tâches : une tâche accède sans limite à sa machine virtuelle
- Le programmeur (et/ou le langage utilisé) doit mettre en place manuellement ces mécanismes de contrôle

Etats de base d'un processus



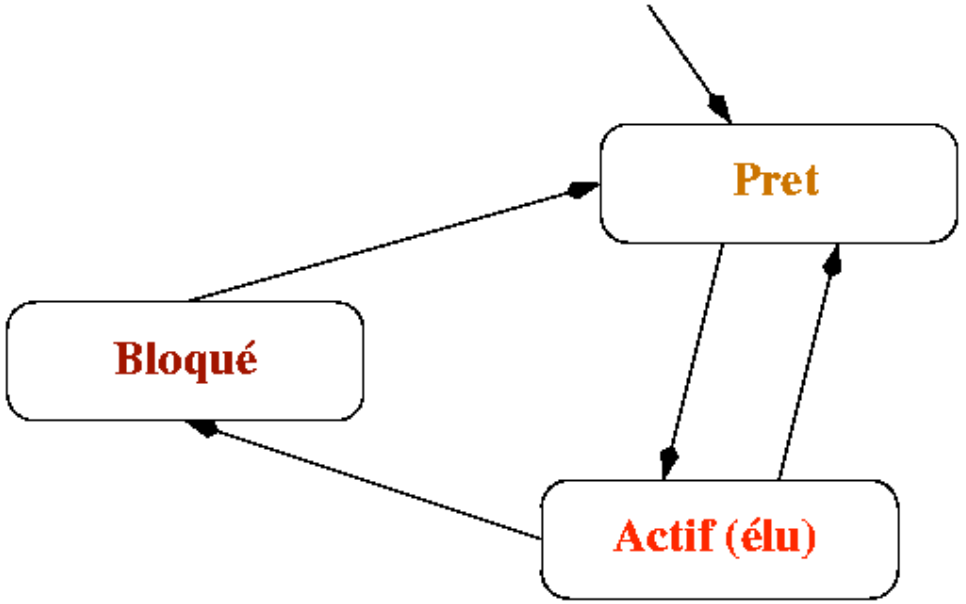
Etat d'initialisation

- L'initialisation d'un processus (ou d'une tâche) consiste à construire un environnement d'exécution

- Selon les constructions offertes par le langage ou le système il est ou non possible de passer en paramètre du processus ou de la tâche des valeurs lui permettant de s'initialiser

Etat d'exécution

L'état "exécution" peut être décomposé en sous états:
prêt/bloqué/exécution réelle



Etat de terminaison

- **Un processus ou une tâche peut terminer sous différentes conditions**
  - fin normale des instructions
  - suicide par l'exécution d'une instruction de terminaison
  - avorté par l'action d'un autre processus/tâche
  - fin sur erreur (exception ou signal non récupéré)
  - lorsque son rôle n'est plus utile
  - jamais

Ecriture et démarrage des processus ou des tâches

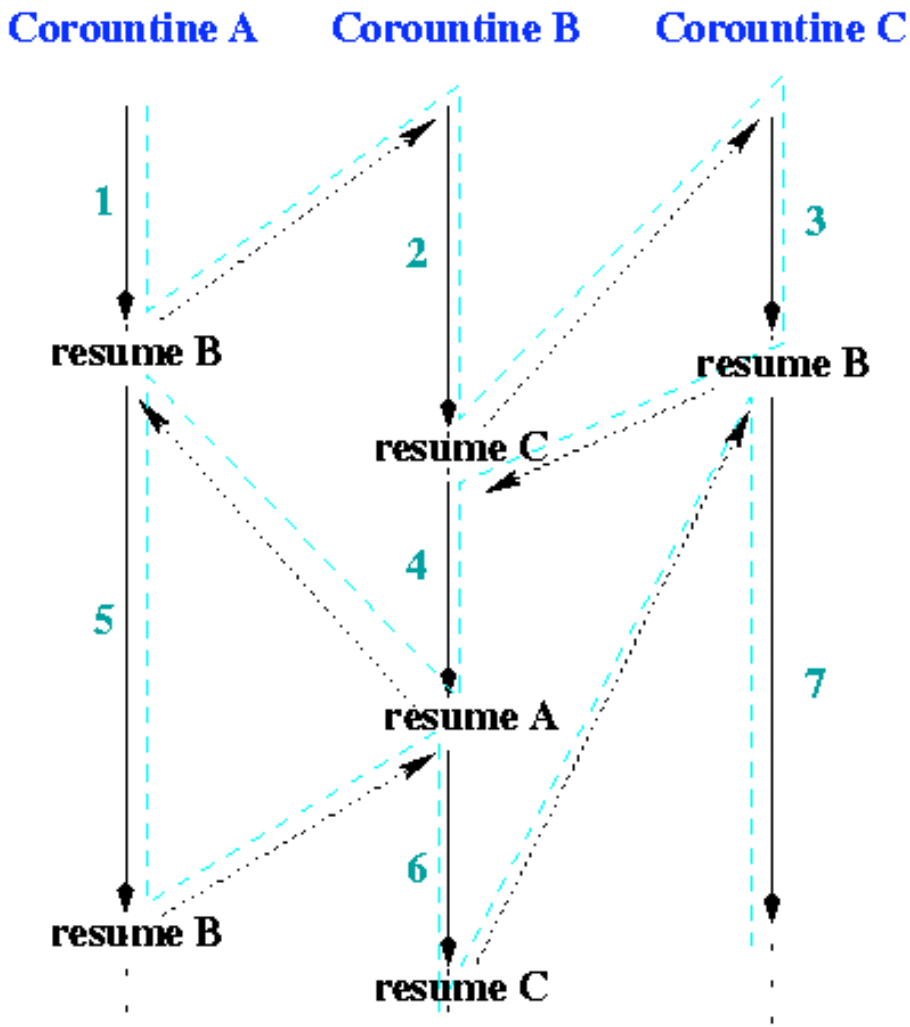
Ecriture

- **Quatre possibilités existent**
  - l'utilisation de coroutines
  - l'utilisation de primitive du type "fork & join »
  - l'utilisation de blocs "cobegin" ou "parbegin »
  - la déclaration explicite des processus avec lancement implicite ou explicite

Les corountines

- **Les coroutines sont des sous-programmes qui se passent le contrôle les unes aux autres par une opération « *resume Nom_Coroutine* »**
- **L'exécution par A d'un « *resume B* » suspend l'activité de la coroutine A au profit de B qui s'exécute alors jusqu'à l'exécution d'un « *resume* ». A reprendra son exécution au point où elle l'avait laissé lorsqu'une coroutine lui passera le contrôle avec un « *resume A* ».**
- **Les données locales à une coroutine sont persistantes d'un appel à l'autre**
- **Chaque coroutine peut être vue comme un processus. Cependant, il n'est pas nécessaire d'utiliser un véritable système concurrent : l'enchaînement des opérations est réalisé par les coroutines elles-même**

Les corountines (suite)



Le « *Fork & Join* »

- La primitive « fork » permet de créer un nouveau processus qui s'exécutera concurremment avec son père

- La primitive « *join* » est l'opération inverse et permet à un père d'attendre la terminaison de ses fils

Le « *Fork & Join* » (suite)

- **L'acceptation classique du *fork* (le fork d'Unix) est que le processus crée ne diffère de son père que par la valeur du retour de cette primitive (0 pour le fils et l'identifiant système du fils pour le père)**

- **Le join en Unix est implémenté par la primitive « *wait* » ou « *waitpid* »**

Voir exemple transparant suivant …/…

Le "Fork & Join" (suite)

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define N 50

int main(void)
{
    pid_t tab_pid[N]; // pid des fils
    int status, i;

    for (i=0; i<N; i++){
        if ( ! (tab_pid[i] = fork()) ){
            // le code du fils; ici un simple affichage et un repos de une seconde
            printf("hello\n");
            sleep(1);
            exit();
        }
    }
    // seul le pere arrive ici car les fils ont termine par exit
    // il attend cette terminaison
    for (i=0; i<N; i++){
        waitpid( tab_pid[i], &status, 0);
    }
    return 0;
}
```

Les blocs « *cobegin* » ou « *parbegin* »

- **Le cobegin est une construction réalisant l'exécution concurente d'un ensemble de séquences d'instructions**

```
cobegin
        Suite_D_Instructions_1
        Suite_D_Instructions_2
        ...
        Suite_D_Instructions_N
coend
```

- **Le bloc termine quand toutes les séquences ont terminées**

Déclaration explicite

- **Certains langages offrent des constructions spécifiques pour la déclaration de tâches ou de processus :**
  - En Ada : type *Task*
  - En Java : classe *thread*

- **L'exécution d'une tâche ou d'un processus est soit faite automatiquement lors du démarrage du programme (cas de Ada) ou faite explicitement par une instruction particulière (cas de Java avec l'appel de la méthode *start()* d'une thread)**

Conclusion

- 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