Cours « système d'exploitation »
2ème année
IUT de Caen, Département d'Informatique
(François Bourdon)

Chapitre 3
Synchronisation de Processus (partie−1)

Plan

3. Synchronisation de Processus

Les machines bi−processeurs existent, les systèmes existent, la difficulté est la programmation elle−même.

On trouve plusieurs types de programmation : séquentielle, structurée, objet, agent, fonctionnelle, parallèle ...

Suivant les langages et les systèmes sous−jacents on manipule les processus de différentes façons.

Un processus peut être considéré comme une occurrence d'un programme, c'est−à−dire comme une entité active/dynamique, contrairement au programme qui est une entité statique.

L'état d'un processus c'est :

- l'état de ses variables globales,
- l'état de la pile,
- l'état du tas,
- l'état de la structure U (Unix),
- l'état du registre du processeur, sauvegardé lors de la commutation de contexte (mot d'état).

3.1 Expression de la notion de processus dans les langages de programmation

Dans Unix : **fork () , exec ()** (PID)

En Pascal// ou en **C//,** il existe une structure de contrôle pour le parallèlisme : cobegin/coend

```
cobegin
        <bloc1> || <bloc2> || <bloc3>
coend
```

à la fin du ***coend*** les 3 blocs sont terminés et les processus 1,2 et 3 sont finis.

Le type « processus » peut être directement manipulable par le langage :

```
process I ( < paramètres formels> )
        var locale;
            begin
              ¦
              ¦
              ¦
              ¦
            end

start  I  (7,  « bonjour »)      ¢ lancement du
                                   processus
```

Dans certains systèmes on manipule une entité plus légère que le processus que l'on appelle un fil d'exécution (***thread***).

Dans ce cas on passe l'adresse de la fonction que l'on veut exécuter sur une ***thread***.

L'interface POSIX des threads est la suivante :

```
#include <pthread.h>
```

```
/* crée un nouveau thread (comme le "fork" pour un
    processus)  */

int pthread_create(pthread_t *thread,
    pthread_attr_t *attr, void * (*lancer_routine)
    (void *), void *arg);

// termine une thread

void pthread_exit (void *retval);

/* attendre la fin d'un thread (comme "wait" pour
    les processus)  */

int pthread_join (pthread_t th, void **thread_return);
```

Il existe des objets processus, qui possèdent des méthodes de manipulation des *threads* :

```
objet o;
thread t (o, m);
t.pause ();
t.resume ();
t.yield ();
t.join (t1); /* rendez−vous */
t.become (o.m2);
```

Il existe également des objets actifs (langages à base d'acteur) :

```
CarrefourProtege   C;    /* classes décrivant */
Voiture            t [100];   /* des objets actifs */

  for (i=0; i<100; i ++)   t [i].run;
```

Dans ce cas on ne parle pas de processus

Enfin on trouve les langages à parallèlisme implicite.

Si l'on considère la recherche d'information dans un arbre binaire :

- en séquentiel tantque le processus courant n'a pas fini sa recherche, un autre processus attend

- en parallèle, plusieurs processus pourront parcourir l'arbre en même temps.

```
class BinTree
        inherit Process
```

3.2 Modèles de représentation des processus

Processus séquentiels

Un processus correspond à une suite d'instructions, issues du programme sous−jacent.

Une tâche est une unité élémentaire de traitement, ayant une cohérence logique.

Si P est un processus, **$P = T_1T_2...T_n$** signifie que le processus **P** est constitué de la suite des exécutions des tâches **$T_1, T_2, ..., T_n$**.

De plus à chaque tâche nous associons deux événements :

d_i : la tâche commence son exécution (d pour début)

f_i : la tâche termine son exécution (f pour fin).

On parle alors de *l'**initialisation*** (lecture des paramètres d'entrée, acquisition de ressources nécessaires, chargement d'information) et de la ***terminaison*** (écriture des résultats, libération des ressources, sauvegarde d'information) d'une tâche.

Définition−1 : A toute suite de tâches $T_1T_2...T_n$ élémentaires est associée une suite $d_1f_1d_2f_2...d_nf_n$ d'initialisations et de terminaisons de tâches.

Définition−2 : On appelle une telle suite, le mot associé à la suite de tâches, écrit sur l'alphabet $A = \{d_1, f_1, d_2, f_2, ... , d_n, f_n\}$.

Exemple :

Soit la tâche T correspondant à l'instruction **N := N + 1**, avec

initialisation (d) : lecture de la valeur de N

interprétation F de T : **F(n) = n + 1**

terminaison (f) : écriture de la nouvelle valeur de N

En utilisant des instructions du langage machine opérant sur un registre **R**, il est possible de décomposer la tâche **T** en trois tâches plus élémentaires :

$$T = T_1 T_2 T_3 \quad \text{et} \quad df = d_1 f_1 d_2 f_2 d_3 f_3$$

où

T_1 correspond à l'instruction	**Load R, N** : chargement de la valeur de N dans le registre R.
T_2 correspond à l'instruction	**Add R, 1** : incrémentation du registre R.
T_3 correspond à l'instruction	**Store R, N** : transfert de la valeur de R dans N.

Systèmes de tâches et graphes de précédence

Dans de tels systèmes les tâches peuvent être exécutées **en parallèles.**

On introduit une relation de **précédence** "<" sur un ensemble **E** de tâches élémentaires :

1. $\forall$ **T** $\in$ **E**, on n'a pas **T < T**,

2. $\forall$ **(T, T') $\in$ E**, on n'a pas simultanément **T < T'** et **T' < T**,

3. la relation "< " est *transitive*, c'est−à−dire que si **T, T' et T'' $\in$ E** tels que **T < T'** et **T' < T''**, alors **T < T''**.

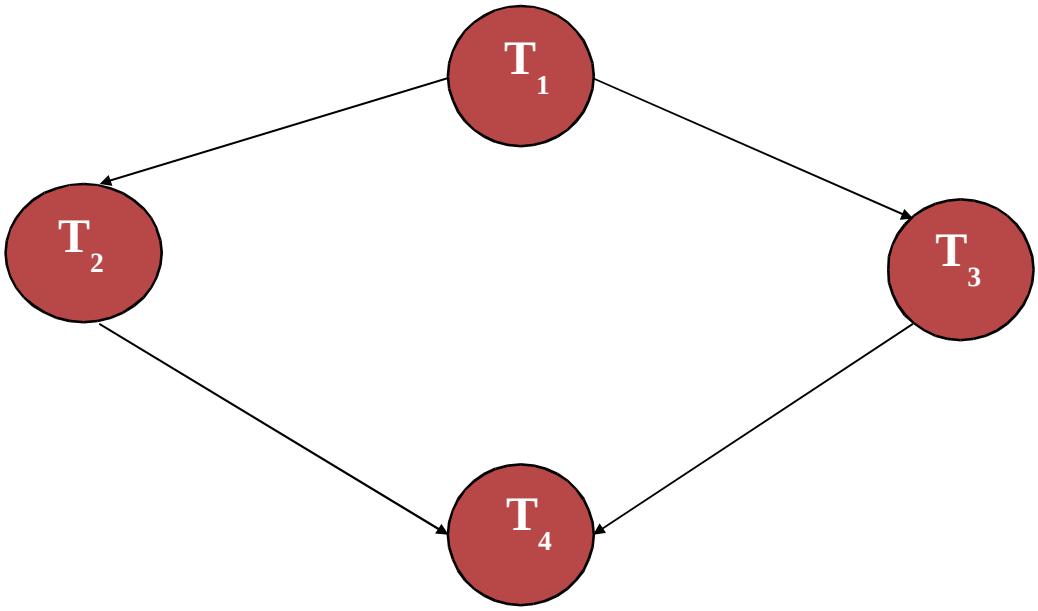
Définition−3 : On appelle **système de tâches S = (E, <)**, un couple constitué d'un ensemble **E** de tâches et d'une relation de précédence "<" sur cet ensemble.

La relation "<" a une interprétation très intuitive :

$T_i < T_j$ ⇔ la terminaison de T_i doit nécessairement intervenir avant l'initialisation de T_j.

Définition−4 : Donc si on n'a ni $T_i < T_j$, ni $T_j < T_i$, c'est que l'ordre dans lequel on effectue T_i et T_j est sans importance. On dira que T_i et T_j sont exécutables en **parallèle**.

Plusieurs mots peuvent satisfaire les contraintes d'un graphe de précédence donné :



C'est le cas pour :

$d_1f_1d_2d_3f_3f_2d_4f_4$,

$d_1f_1d_3d_2f_2f_3d_4f_4$ ou

$d_1f_1d_3f_3d_2f_2d_4f_4$.

Définition−5 : L'ensemble de ces mots décrivent tous les comportements possibles du système de tâches **S**, on l'appelle le **langage L(S)** de **S**. Un mot **w** de **L(S)** correspond à un comportement de **S**.

On pourra faire des opérations de **produit** et de **composition parallèle** de graphes **G** et **G'** de systèmes de tâches.

Définition−6 : Le produit $G_1 * G_2$ est le graphe de précédence obtenu à partir des graphes G_1 et G_2, en ajoutant des arêtes joignant chaque sommet terminal de G_1 à chaque sommet initial de G_2.

Dans l'interprétation temporelle introduite précédemment, cela signifie que $G_1 * G_2$ reprend les contraintes de G_1 et de G_2, avec en plus la contrainte qu'aucune tâche de G_2 ne peut être initialisée avant que toutes les tâches de G_1 ne soient terminées.

Spécification dans les langages évolués

Certains langages permettent d'exprimer la mise en séquence et en parallèle de systèmes de tâches.

- la mise en séquence :

I_1 ; I_2 ;	pour des instructions élémentaires.
P_1 ; P_2 ;	pour des programmes associés à des systèmes de tâches **S_1** et **S_2**.

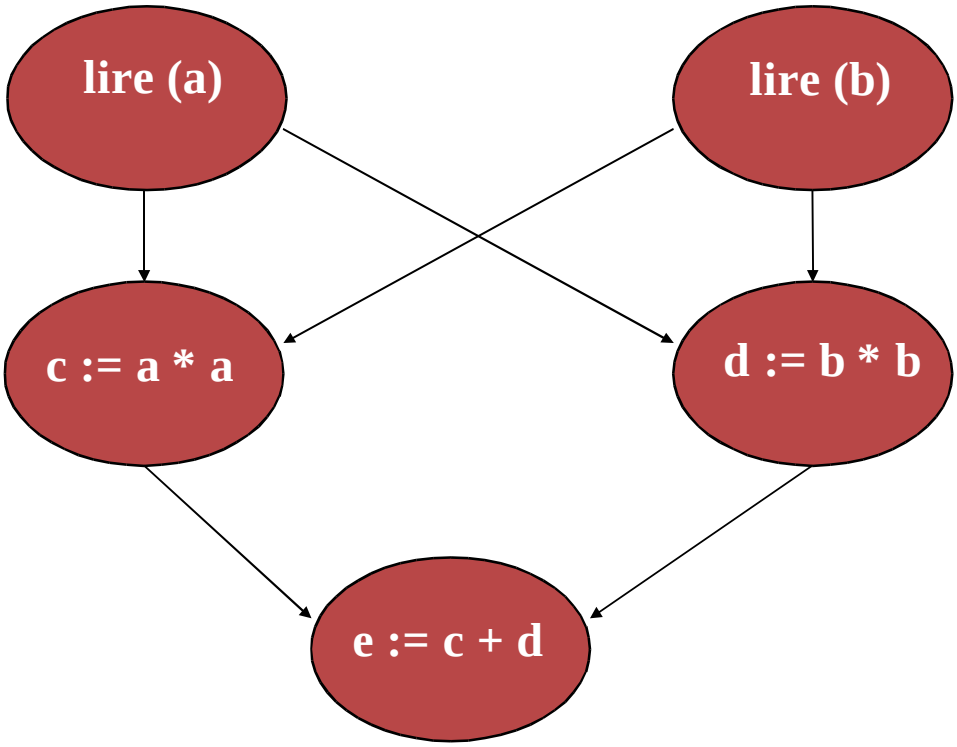
- la mise en parallèle :

parbegin I_1 ; I_2 ; ... ; I_n parend ;	pour les instructions
parbegin P_1 ; P_2 ; ... ; P_n parend ;	pour les programmes

Le programme

```
début
        parbegin lire(a); lire(b) parend;
        parbegin c:=a*a; d:=b*b parend;
        e:=c+d;
fin.
```

correspond au graphe de précédence



Il existe d'autres modèles pour décrire un ensemble de processus et analyser leurs comportements. Les plus courants sont les **automates** et les **réseaux de Petri.**

Les Automates finis

Ils sont adaptés à des situations pouvant être décrites par une **évolution d'état en état, provoquée par des actions**.

> *Exemple* : Un processus séquentiel ayant un nombre fini de variables entières à valeurs bornées, peut être considéré comme évoluant d'état en état, sous l'effet d'instructions élémentaires.

Un état est ici le contexte du processus, incluant la suite des valeurs des variables et le compteur ordinal. Puisque les variables sont bornées, il y a un **nombre fini d'états possibles**. Ainsi même si le processus est infini, son comportement peut être capturé par une structure finie.

Un automate fini est un quadruplet $\mathcal{A} = <\sum, E, \delta, e_0>$, où

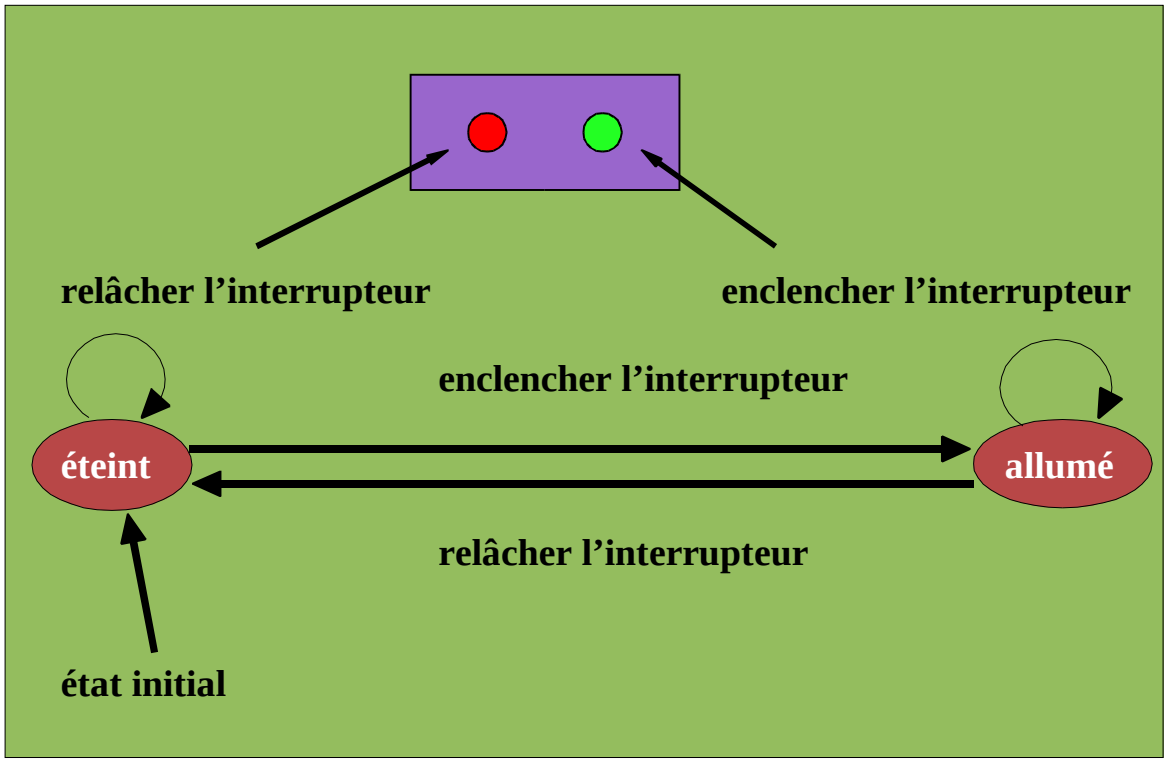
$\sum$ est un **ensemble fini d'actions,**

E est un **ensemble fini d'états,**

e_0 est un élément particulier de **E**, appelé **état initial,**

δ est une application de $E \times \sum \rightarrow E$, appelée **fonction de transition**.

La *fonction de transition* indique, lorsque l'automate est dans un certain état et qu'une certaine action se produit, le nouvel état dans lequel il passe.



Automate fini représentant l'état d'une ampoule électrique, les actions étant des mouvements de l'interrupteur. On a ici :

E = { allumé, éteint }

A = { enclencher l'interrupteur, relacher l'interrupteur }

$\mathbf{e_0}$ = éteint (par exemple)

δ (éteint, enclencher l'interrupteur) = allumé

δ (éteint, relâcher l'interrupteur) = éteint

δ (allumé, relâcher l'interrupteur) = éteint

δ (allumé, enclencher l'interrupteur) = allumé

Les Réseaux de Petri

Les réseaux de Petri permettent de représenter, toujours par une structure finie, des situations qui se décrivent en termes **d'évènements et de conditions**.

Exemples : Si l'imprimante est libre (condition), elle est affectée à un certain processus (événement) ; si tous les processus fils sont terminés, le processus père peut continuer ; si son niveau est masqué, l'interruption n'est pas prise en compte, sinon la commutation de contexte intervient et les niveaux inférieurs sont masqués ...

Un réseau de Petri est un triplet $\mathcal{R} = < P, T, \Phi >$ où :

$P = \{p_1, ..., p_n\}$ est un **ensemble fini de places**,

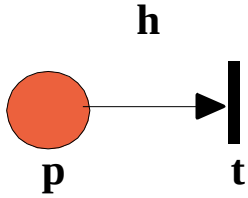
$T = \{t_1, ..., t_n\}$ est un **ensemble fini de transitions**, disjoint de **P**,

$\Phi : (P \times T) \cup (T \times P) \rightarrow N$ (ensemble des entiers naturels) est une **fonction de valuation**.

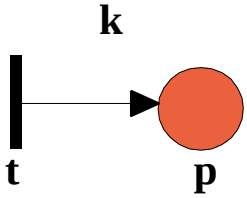
Sur la représentation graphique les places sont figurées par des cercles et les transitions par des barres verticales. Les flèches relient les **places** aux **transitions** (et réciproquement). Les flèches sont étiquetées par les entiers définis par la **fonction de valuation**.

Une flèche de valuation **h** relie la place **p** à la transition **t**, si :

$$\Phi(p, t) = h$$



place d'entrée



place de sortie

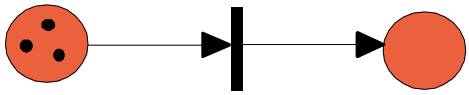
Chaque place peut contenir un nombre illilmité de jetons. Au départ on distribue de façon quelconque des jetons dans des places, c'est le **marquage initial**.

Chaque distribution de jetons dans les places s'appelle un **marquage** du réseau. Pour un marquage donné, certaines transitions sont **franchissables**, d'autres pas.

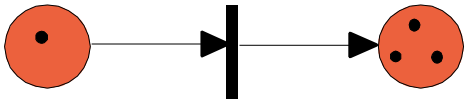
Définition−7 : Une transition est franchissable si chacune de ses places d'entrées contient un nombre de jetons au moins égal à sa valuation (en tant que place d'entrée de la transition).

Définition−8 : Le franchissement d'une transition enlève dans chaque place d'entrée un nombre de jetons correspondant à sa valuation et ajoute à chaque place de sortie un nombre de jetons égal à sa valuation (en tant que place de sortie de la transition).

Situation avant franchissement :

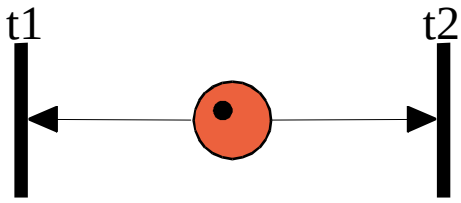


Situation après franchissement de la transition :



Un jeton ne peut pas être utilisé pour franchir deux transitions différentes.

Sur l'exemple ci−après les deux transitions sont franchissables, mais seule l'une d'entre elles peut être franchie.



Transitions en exclusion mutuelle

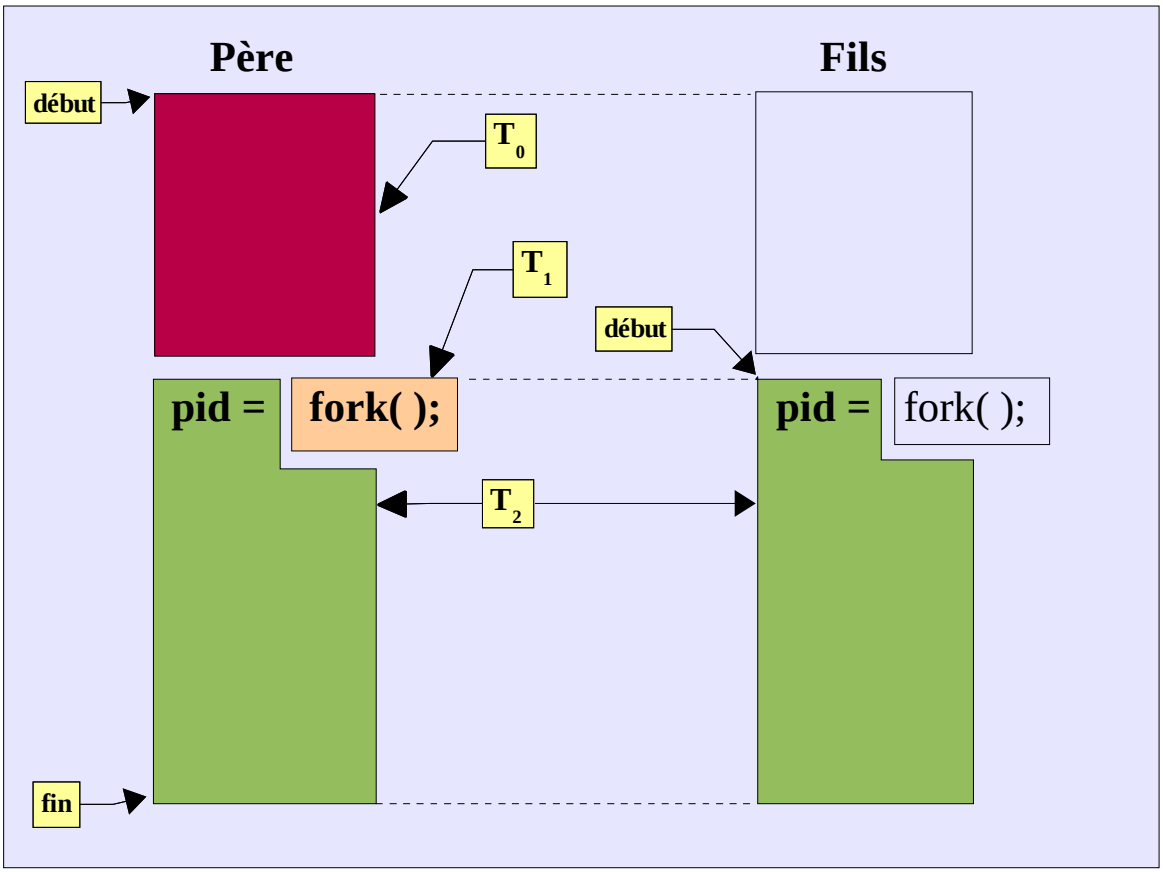
Reprenons les graphes de tâches et regardons comment exprimer le parallélisme implicite dans UNIX.

Exemple de mise en oeuvre sur UNIX

Il existe deux façons dans UNIX d'exprimer du parallélisme :

- création de processus par l'appel "**fork ( )**", ou
- utilisation des "**threads**"

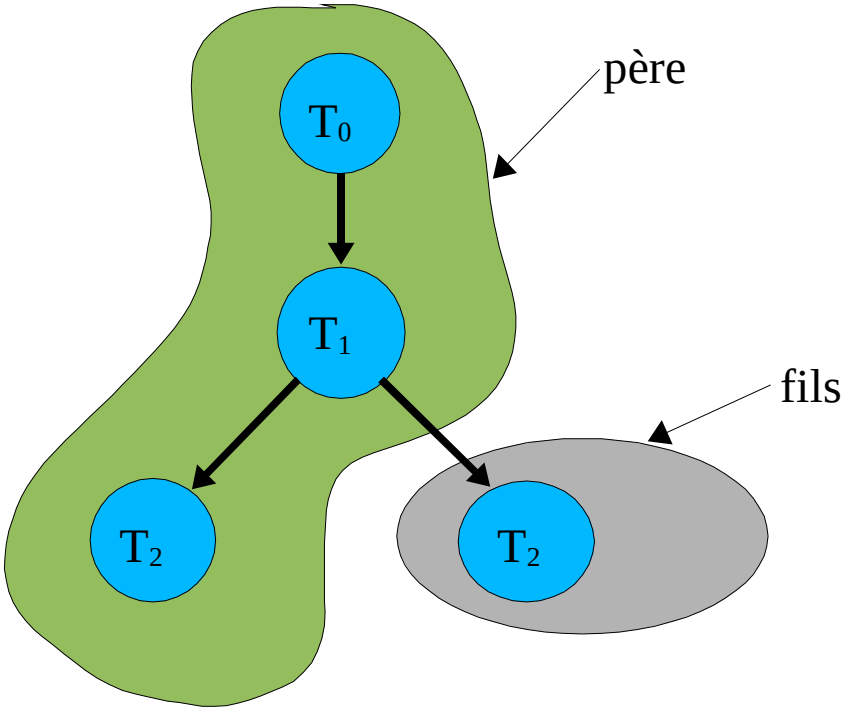
Regardons comment l'utilisation d'un appel à la fonction "**fork**" peut être exprimée sous la forme d'un graphe de tâches :



Cela s'écrit en pseudo−code :

```
début
        T0 ;
        T1 ;
         par_begin  T2 ; T2  par_end ;
fin ;
```

D'où le graphe de tâches correspondant :



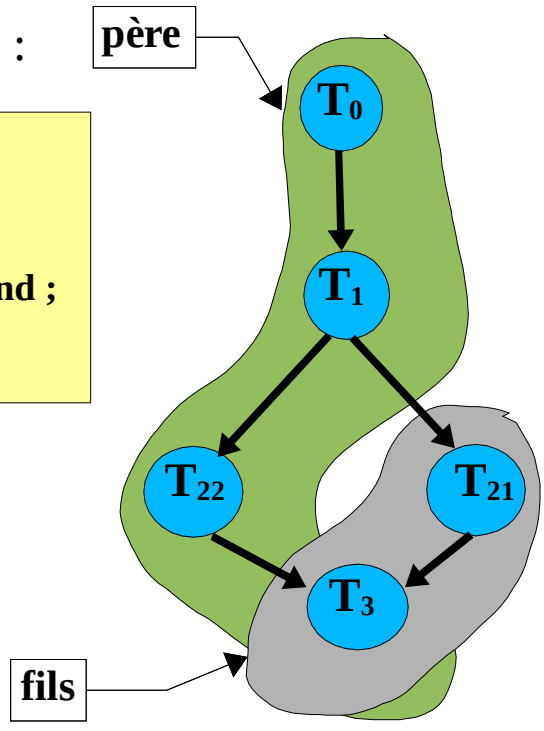
L'utilisation classique de la fonction **fork** est la suivante :

```
pid=fork();

if (pid==0) { /* code correspondant à
              l'exécution du processus fils
              */ }

else { /* code correspondant à
          l'exécution du processus
          père */ }
```

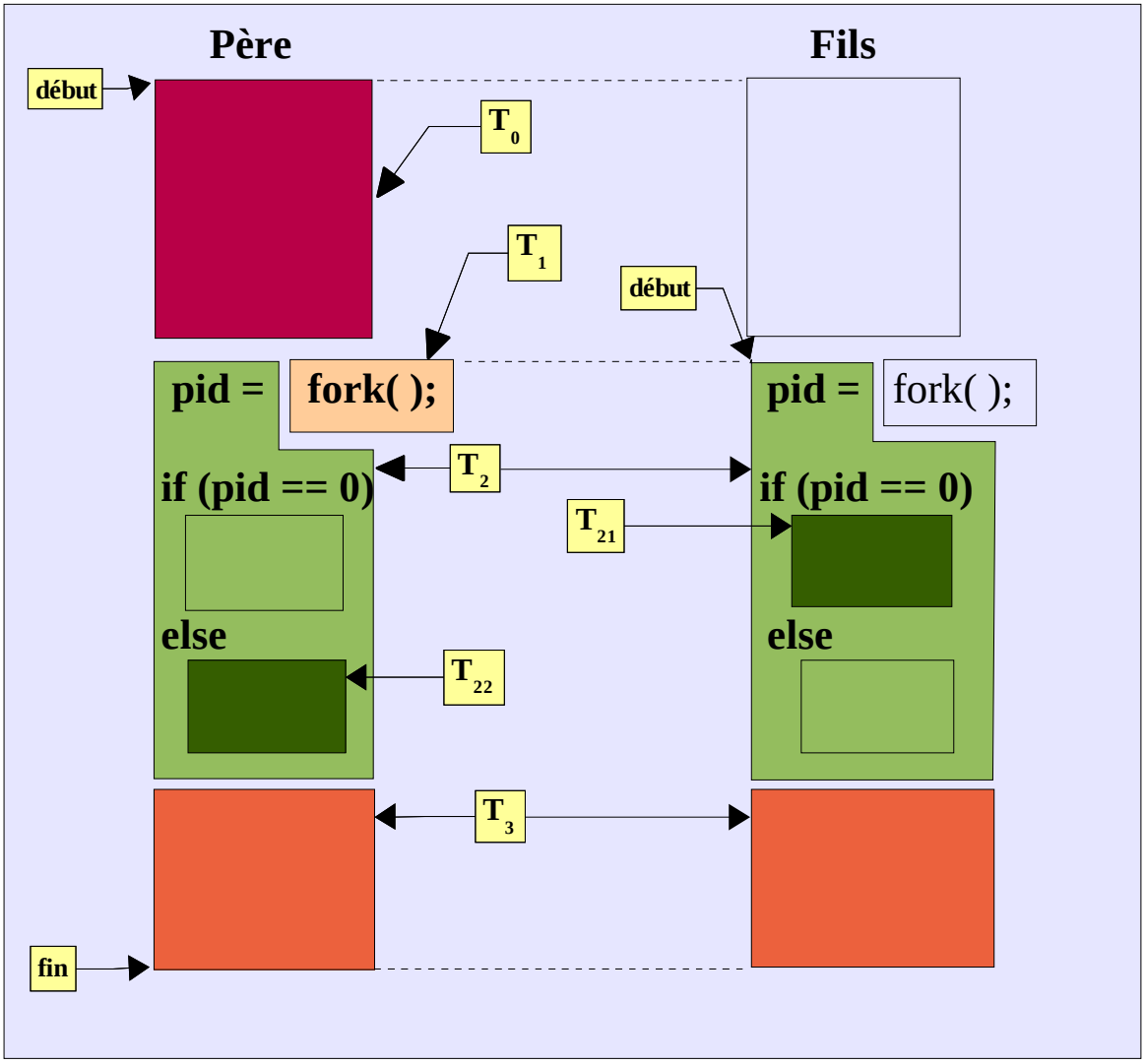
Dans ce cas la tâche T_2 se décompose en deux tâches T_{21} et T_{22} ; il apparaît ensuite la tâche T_3 commune aux deux processus, car après le "*if then else*".

Cela s'écrit en pseudo−code :

```
début
        T0 ;
        T1 ;
        par_begin T22 ; T21 par_end ;
        T3 ;
fin ;
```



Regardons comment l'utilisation de l'appel à la fonction "**fork**" est réalisée dans ce cas "classique" :



Prenons un exemple plus complet dans UNIX :

i) Un processus crée un fils.

ii) Il arme le signal **SIGUSR1** sur un "handler" donné.

iii) Le fils émet le signal correspondant à son père.

=> des contraintes de synchronisation/parallélisme

1° : ii < iii
2° : Le fils doit se terminer avant le père

Nous avons vu que T_{21} et T_{22} sont les comportements respectifs du fils et du père après le "**fork**".

T_{22} : le père attend le signal du fils et se synchronise sur sa fin.

T_{21} : le fils émet un signal à son père et se termine.

Pour réaliser cela nous disposons dans UNIX des primitives suivantes :

signal() : armement d'un signal
fork() : création d'un processus
pause() : mise en sommeil sur l'arrivée d'un signal
wait() : mise en sommeil sur la terminaison d'un fils
sleep() : mise en sommeil sur une durée déterminée (argument)
kill() : envoi de signal
exit() : terminaison d'un processus

En TP nous voyons comment utiliser ces primitives pour obtenir le comportement "père−fils" souhaité.

Les comportements des deux processus, décrits de façon séquentielle, sont les suivants :

Père :

```
signal( ) ;      // t0
fork( ) ;        // t1
pause( ) ;       // t2
wait( ) ;        // t3
printf( ) ;      // t4
```

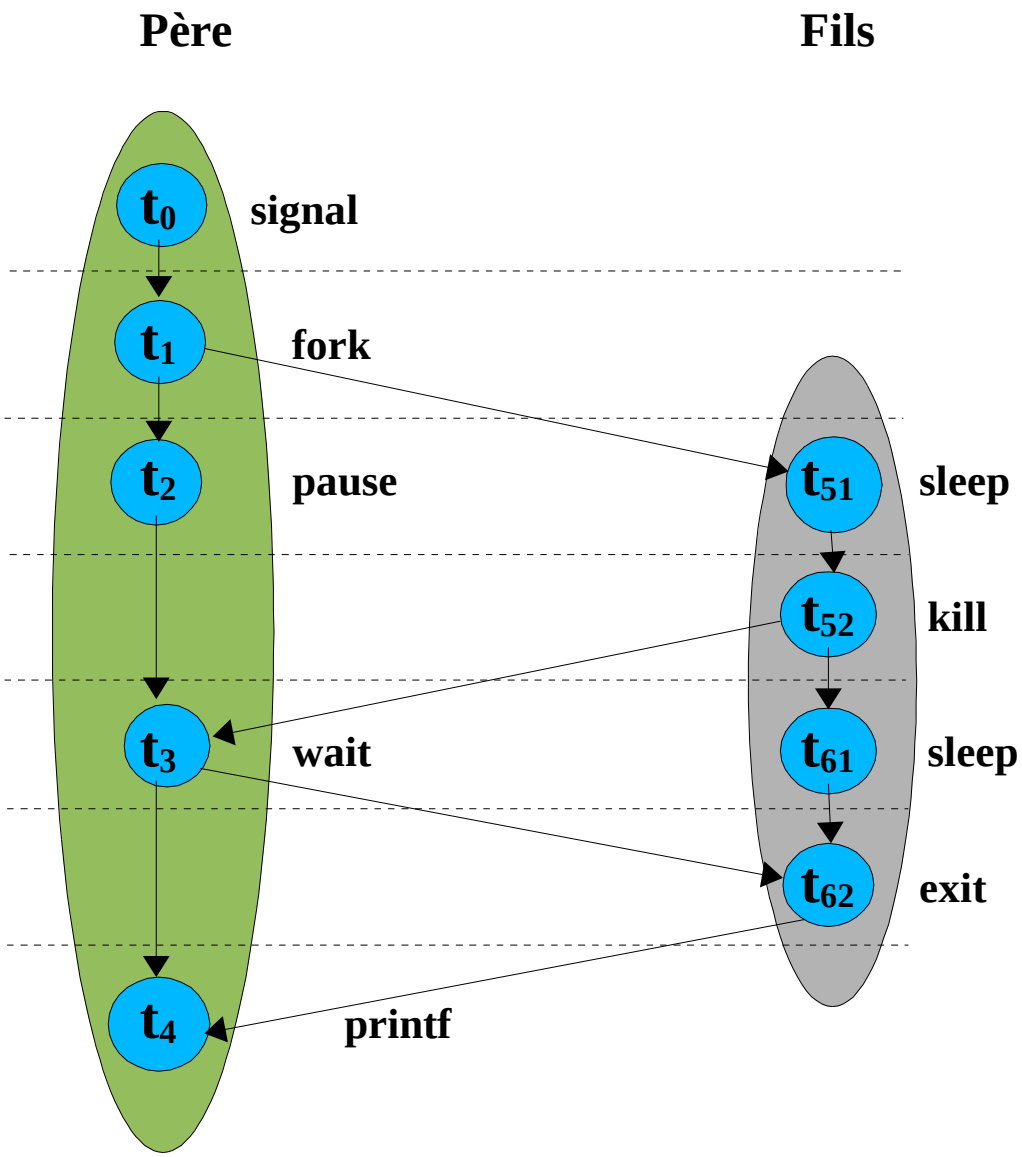
Fils :

```
sleep( ) ;  // t51
kill( ) ;   // t52
sleep( ) ;  // t61
exit( ) ;   // t62
```

Une synchronisation forte s'exprime de façon explicite (pseudo_code) comme ceci :

```
début :
                    t0 ;
                    t1 ;
    par_begin       t2 ;        t51     par_end
                                t52 ;
    par_begin       t3 ;        t61     par_end
                                t62 ;
                    t4 ;

fin ;
```

Graphe des tâches correspondant



L’autre technique sur UNIX pour faire du parallélisme est d’utiliser les "threads".

RAPPEL_1 : A l’exécution d’un appel **fork** par un processus, une nouvelle copie de celui−ci est créée avec ses propres variables et son propre **PID**. Ce nouveau processus est indépendamment ordonnancé et s’exécute quasi indépendamment du processus l’ayant créé.

DIFFERENCE_1 : Lors de la création d’un nouveau thread dans un processus, il obtient sa propre pile (et de ce fait ses propres variables locales) mais partage avec son créateur les variables globales, les descripteurs de fichiers, les fonctions de gestion des signaux et l’état du répertoire courant. Linux utilise l’appel système **clone** pour créer des threads.

Créer un nouveau thread nécessite de passer en argument à la méthode "**pthread_create**" l'adresse d'une fonction et l'argument unique lui devant être transmis.

DIFFERENCE_2 : L'utilisation de **fork** génère la poursuite de l'exécution au même emplacement avec un code de retour différent, **tandis qu'un nouveau thread commence son exécution avec la fonction passée en paramètre**.

Exemple_1 : Ce programme crée un autre thread qui va montrer qu'il partage des variables avec le thread original, et permettre au "petit nouveau" de renvoyer un résultat à l'original.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *fonction_de_thread(void *arg);

char message[] = "Hello World";

int main() {
  int res;
  pthread_t un_thread;
  void *resultat_de_thread;
  res=pthread_create(&un_thread, NULL, fonction_de_thread, (void
*)message);
  if (res != 0) {
    perror("Echec de la creation du thread ");
    exit(EXIT_FAILURE);
  }
  printf("En attente de terminaison du thread ...\n");
  res=pthread_join (un_thread, &resultat_de_thread);
  if (res != 0) {
    perror("Echec de l'ajout de thread ");
    exit(EXIT_FAILURE);}
  printf("Retour du thread, il a renvoye %s\n", (char *) resultat_de_thread);
  printf("Voici a present le message %s\n", message);
  exit(EXIT_SUCCESS);
}

void * fonction_de_thread (void *arg) {
  printf("la fonction_de_thread est en cours d'execution. L'argument etait
%s\n", (char *) arg);
  sleep (3);
  strcpy(message, "Salut !");
  pthread_exit("Merci pour le temps processeur");}
```

adresse de l'identifiant du thread créé

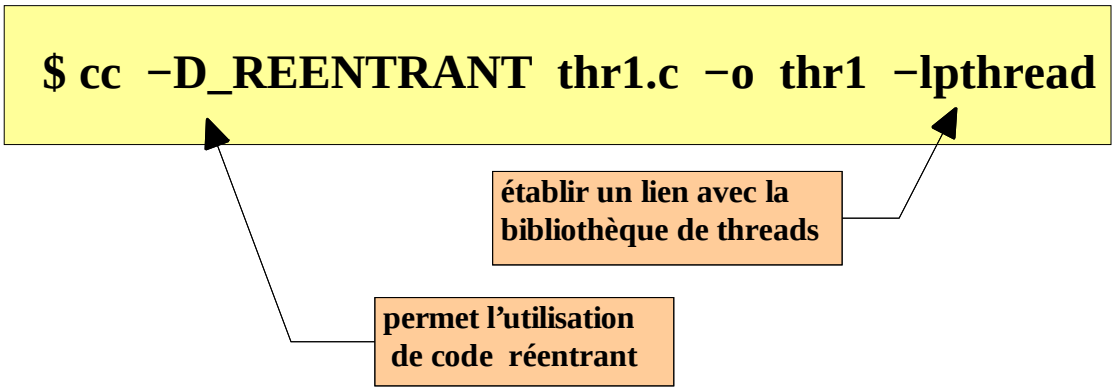
fonction exécutée par le thread créé

paramètre passé à la fonction

identifiant du thread attendu

adresse de l'objet renvoyé au thread appelant

Compilation de l'*exemple_1* :



Résultat de l'exécution :

```
$ ./thr1
En attente de terminaison du thread ...
la fonction_de_thread est en cours d'exécution.
L'argument était Hello World
Retour du thread, il a renvoyé Merci pour le temps processeur
Voici à présent le message Salut !
$
```

thread appelé

thread appelant