Processus concurrents et parallélisme

Chapitre 2 - Concurrence et Parallélisme

Gabriel Girard

17 octobre 2022

Chapitre 2 - Concurrence et Parallélisme

Chapitre 2 - Concurrence et Parallélisme

Définition

Définition

- Deux processus sont concurrents s'ils existent en même temps

Définition

- Deux processus sont concurrents s'ils existent en même temps

Pas de problème si asynchrones

$s = f(e)$

Définition

- Deux processus sont concurrents s'ils existent en même temps

Pas de problème si asynchrones

$s = f(e)$

Des problèmes se produisent s'ils sont synchrones

$s = f(e, t)$

Définition

- Deux processus sont concurrents s'ils existent en même temps

Pas de problème si asynchrones

$s = f(e)$

Des problèmes se produisent s'ils sont synchrones

$s = f(e, t)$

Les processus doivent se synchroniser et coopérer

Définition : condition de course

Une situation où plusieurs processus accèdent simultanément à la même ressource et que le résultat final dépend de l'ordre d'exécution s'appelle une **condition de course**.

Chapitre 2 - Concurrence et Parallélisme

Il existe deux types de concurrence :

- Concurrence implicite (Unix, Windows,...)
  - parallélisme indépendant
  - granularité très grossière à grossière (2000 à 1M d'instructions)
- Concurrence explicite (langage ou matériel spécialisé)
  - granularité moyenne (20 à 200 instructions)
  - granularité fine (moins que 20 instructions)

Pseudo-parallélisme vs parallélisme

- Entrelacés (un processeur)
- Multiprocesseurs

Taxonomie du parallélisme

Taxonomie de Flynn

	Une instruction	Plusieurs instructions
Une donnée	SISD Architecture von Neumann	MISD Rare.. systolic array ?, pipeline ?, tolérance aux fautes ?
Plusieurs données	SIMD CM-1/CM-1, Sparc VIS, Intel MMX, Power Altivec, Nvidia CUDA, ...	MIMD Multi-processeurs/coeurs, grappe, grid, ...

Chapitre 2 - Concurrence et Parallélisme

Définition

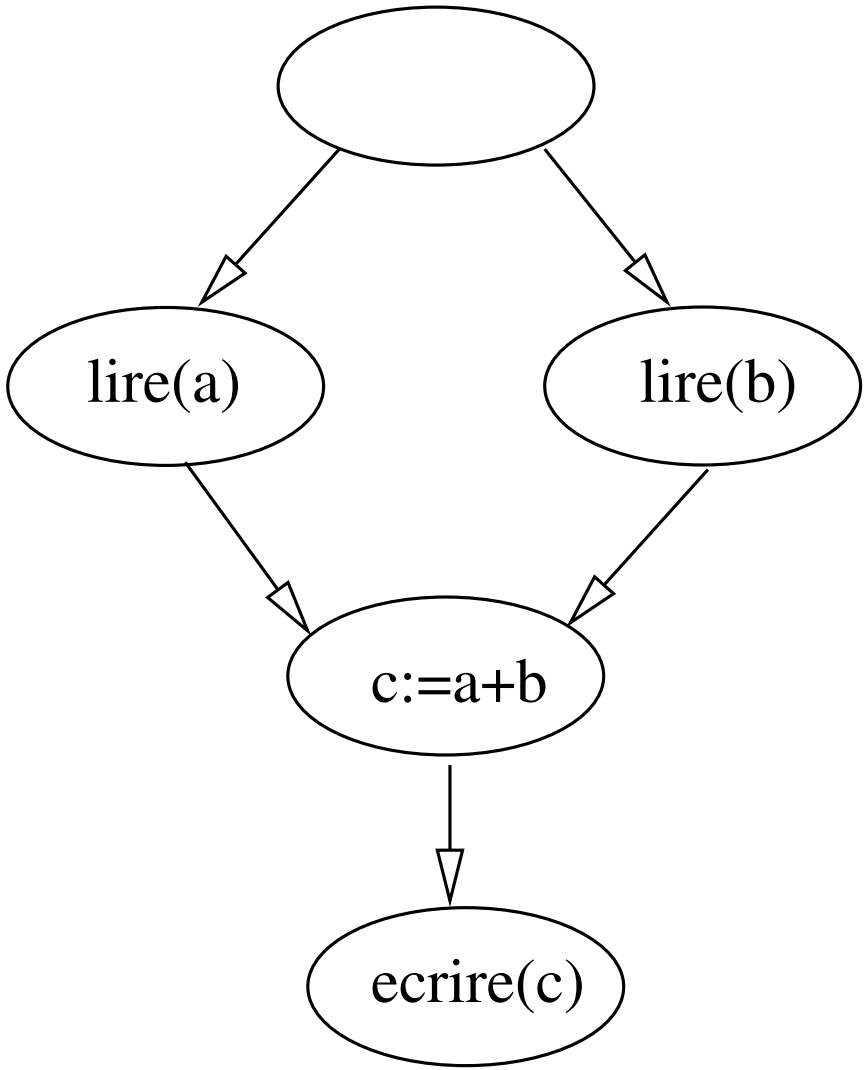
Un graphe de précédence est un graphe acyclique

- noeuds = énoncés
- arc = ordre d'exécution

Exemple

```
lire(a);
lire(b);
c := a + b;
ecrire(c);
```

Exemple



- Approches formelles
  - Logique temporelle
  - Réseau de Petri
  - Ordres partiels
  - Graphes d'événements
  - ...

Chapitre 2 - Concurrence et Parallélisme

Au moins quatre opérations distinctes

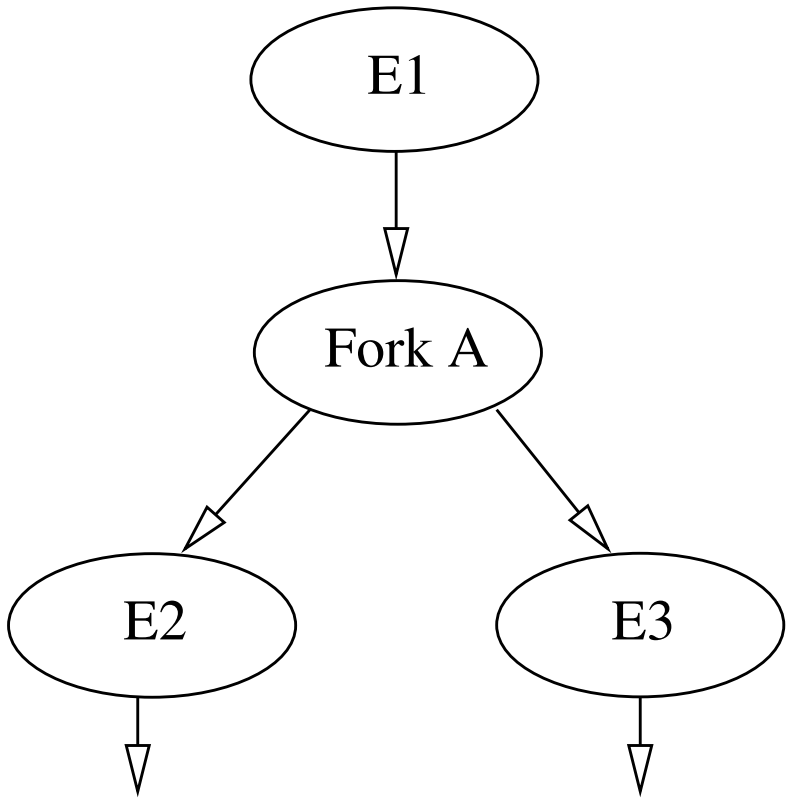
- Fork/join
- Cobegin/coend
- Coroutines
- Déclaration explicite de processus

Versions

- Deux formes de fork/join
- Introduite en 1963 et 1966

Version 1

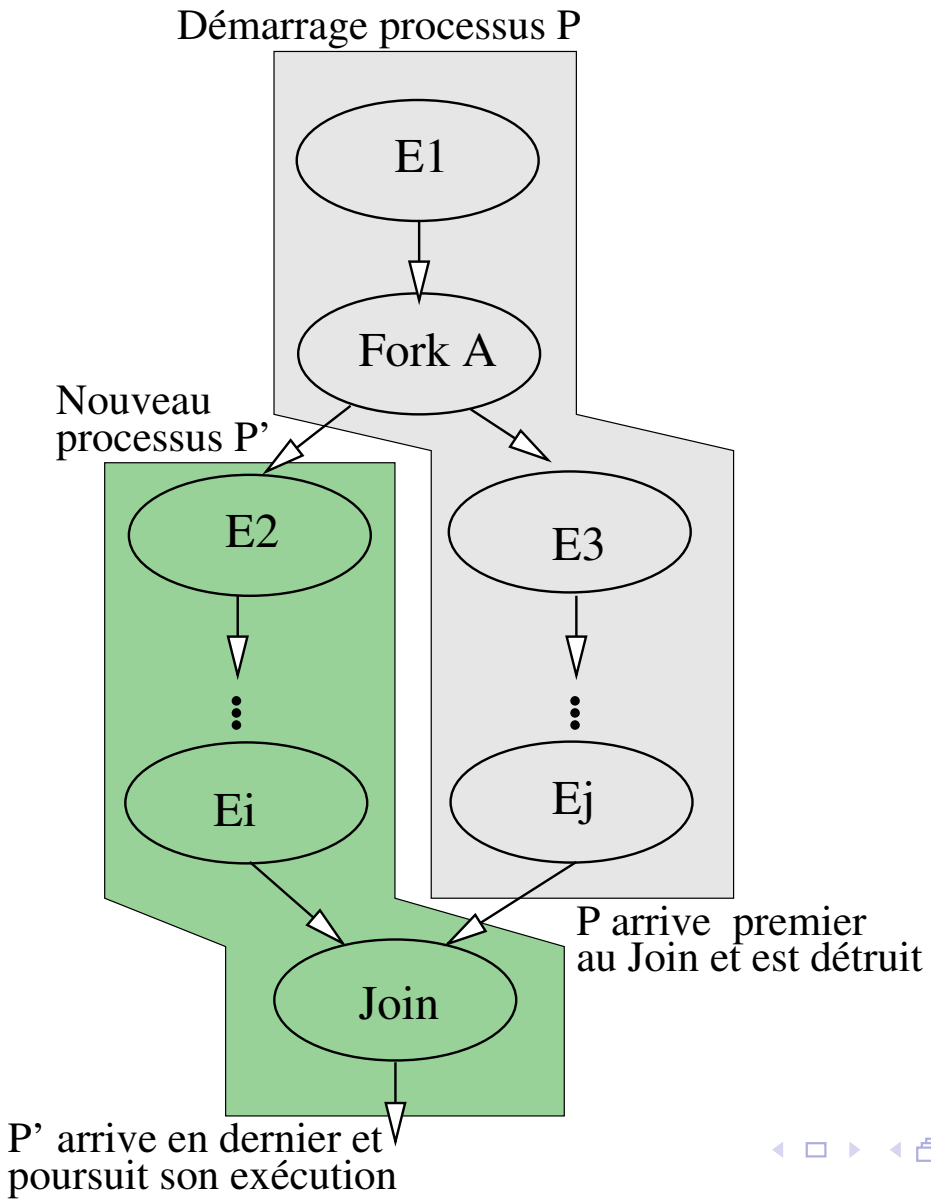
```
        E1
        Fork A
        E2
        ...
A:      E3
        ...
```



Version 1

- `Join` permet de combiner deux traitements parallèles
- Les deux processus exécutent le `join`
- Le premier à l'exécuter termine
- Le second poursuit son exécution

Version 1


Démarrage processus P
E1
Fork A
Nouveau
processus P'
E2
E3
Ei
Ej
P arrive premier
au Join et est détruit
Join
P' arrive en dernier et
poursuit son exécution

Version 1

- Si plus de deux traitements
- On ajoute un paramètre : le nombre de traitements
- Tous les processus sauf le dernier terminent lors du `join`

Version 1

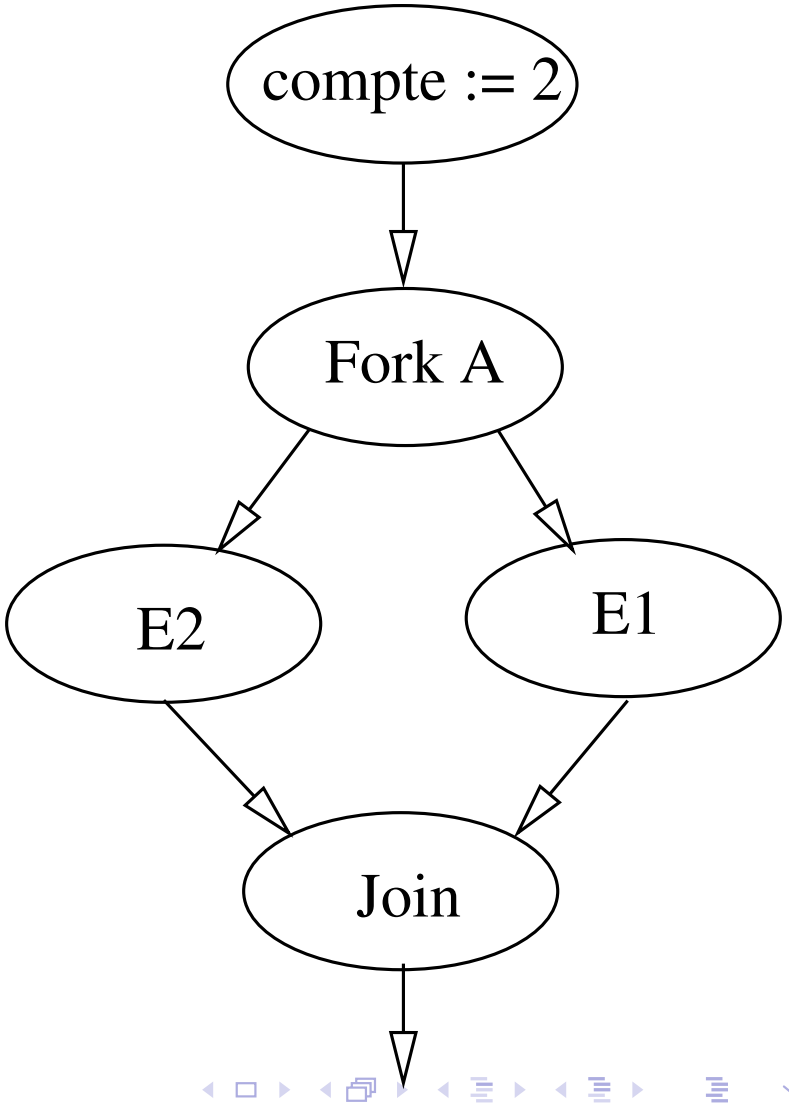
- `Join compte` $\Rightarrow$

```
compte:=compte -1;
if (compte != 0) then quit
```

- L'opération doit être atomique

Version 1

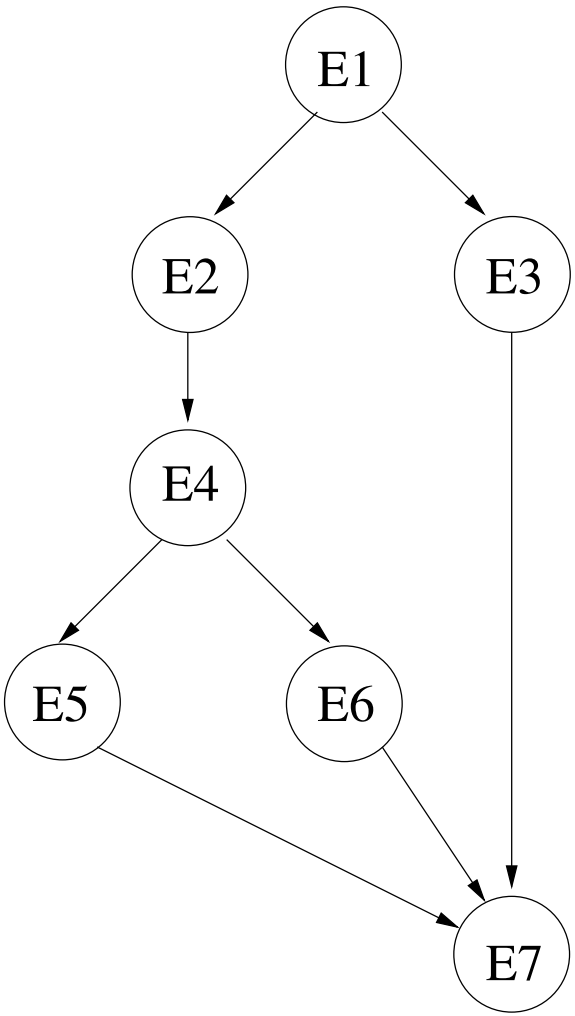
```
        compte := 2
        Fork  A
        ...
        E1
        goto  B
A:      E2
B:      join  compte
        ...
```



Version 1

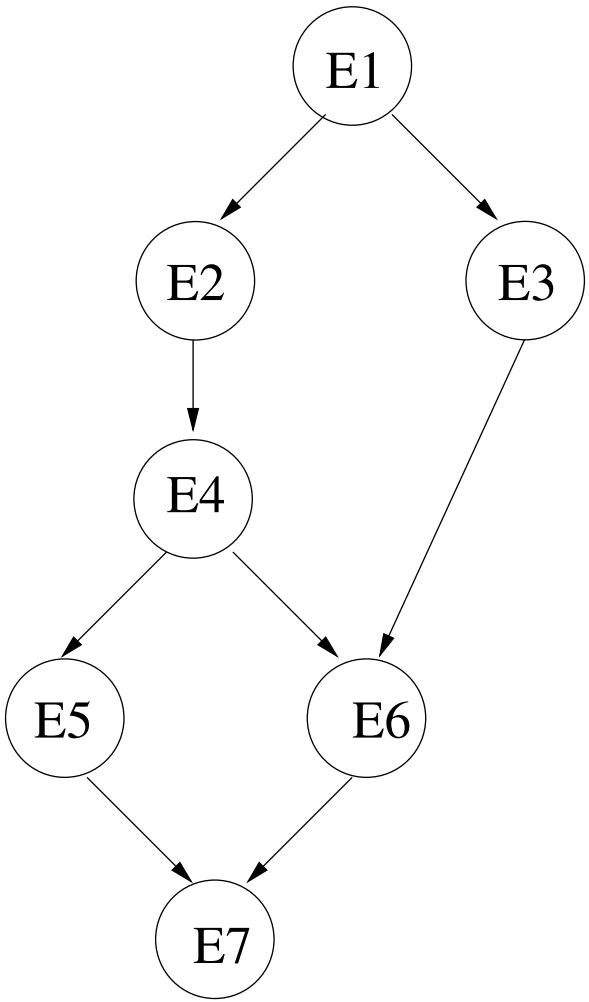
```
lire(a);
lire(b);
c := a + b;
ecrire(c);
```

Version 1




```
     E1;
     compte := 3;
     fork L1;
     E2;
     E4;
     fork L2;
     E5;
     goto L3;
L2:  E6;
     goto L3;
L1:  E3;
L3:  join compte;
     E7;
```

Version 1




```
        E1;
        compte1 = 2;
        fork  L1;
        E2;
        E4;
        compte2 = 2;
        fork  L2;
        E5;
        goto  L3;
L1:     E3;
L2:     join  compte1;
        E6;
L3:     join  compte2;
        E7;
```

Version 1

```
var  f,g : file of T;
     r,s : T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      begin
         write(g,r);
         read(f,r);
      end;
   write(g,r);
end.
```

Version 1

```
var  f,g : file of T;
     r,s : T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      begin
         s := r;
         write(g,s);
         read(f,r);
      end;
   write(g,r);
end.
```

Version 1

```
Var   f,g : file of T;
      r,s : T;
      cpt : integer;
  begin
      reset(f);
      read(f,r);
      while not eof(f)
         begin
              cpt := 2;
              s := r;
              fork L1;
              write(g,s);
              goto  L2;
          L1: read(f,r);
          L2: join  cpt;
          end;
      write(g,r);
  end.
```

Version 2

- Un seconde implantation ressemble à un appel de procédure asynchrone
- Le `Fork` et le `Join` spécifie une fonction en paramètre
- Le `Join` bloque l'émetteur jusqu'à la fin de la procédure asynchrone

Version 2

```
int main()
   ...
   Fork Fct2();
   ...
   Join Fct2()
   ...
```

```
int Fct2()
   ...
   ...
   end;
```

Version 2

DANGER....

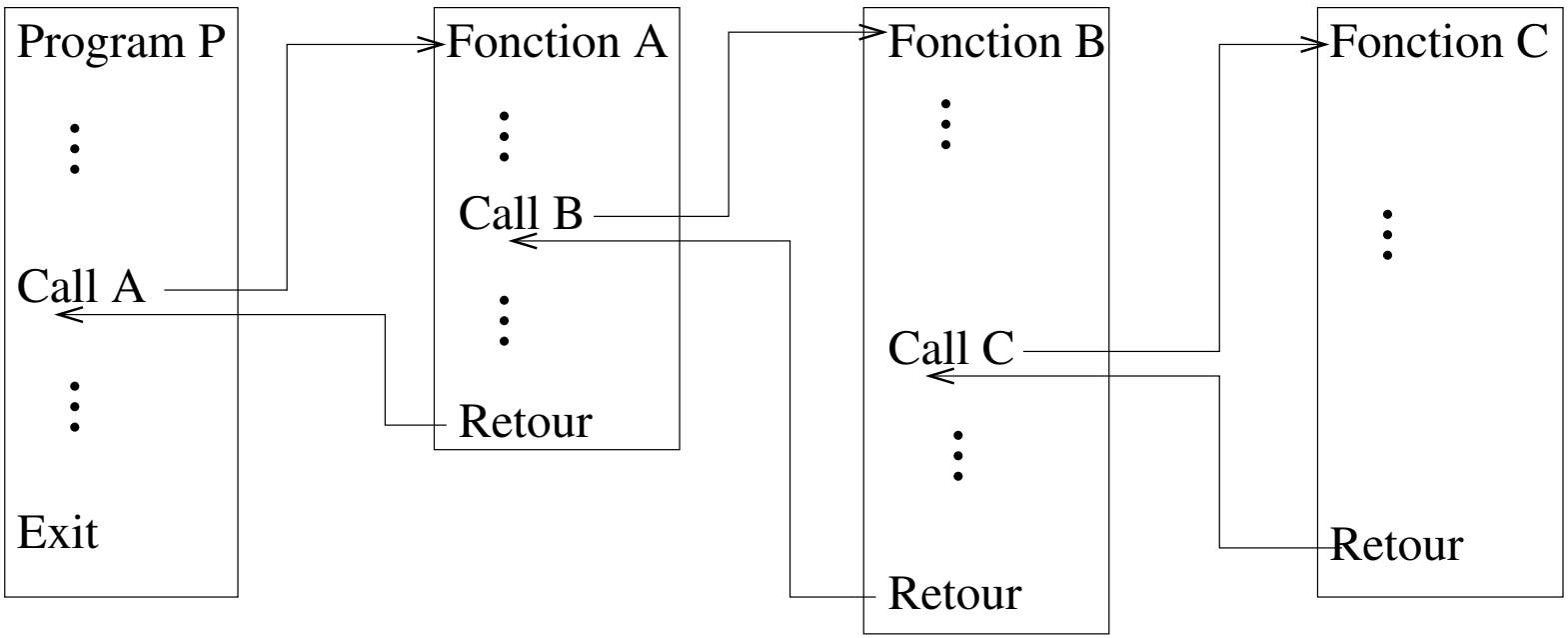
Boucle infinie de création

- Conclusion : dangereuses mais pratiques et puissantes

Définition

- Semblable à des sous-routines (procédures ou fonctions)
- Transfert de contrôle symétrique plutôt que hiérarchique
- Transfert de contrôle : `resume`
- `resume` est la seule technique de transfert de contrôle (appel et retour)
- Peut se faire partout (pas seulement à la fin)

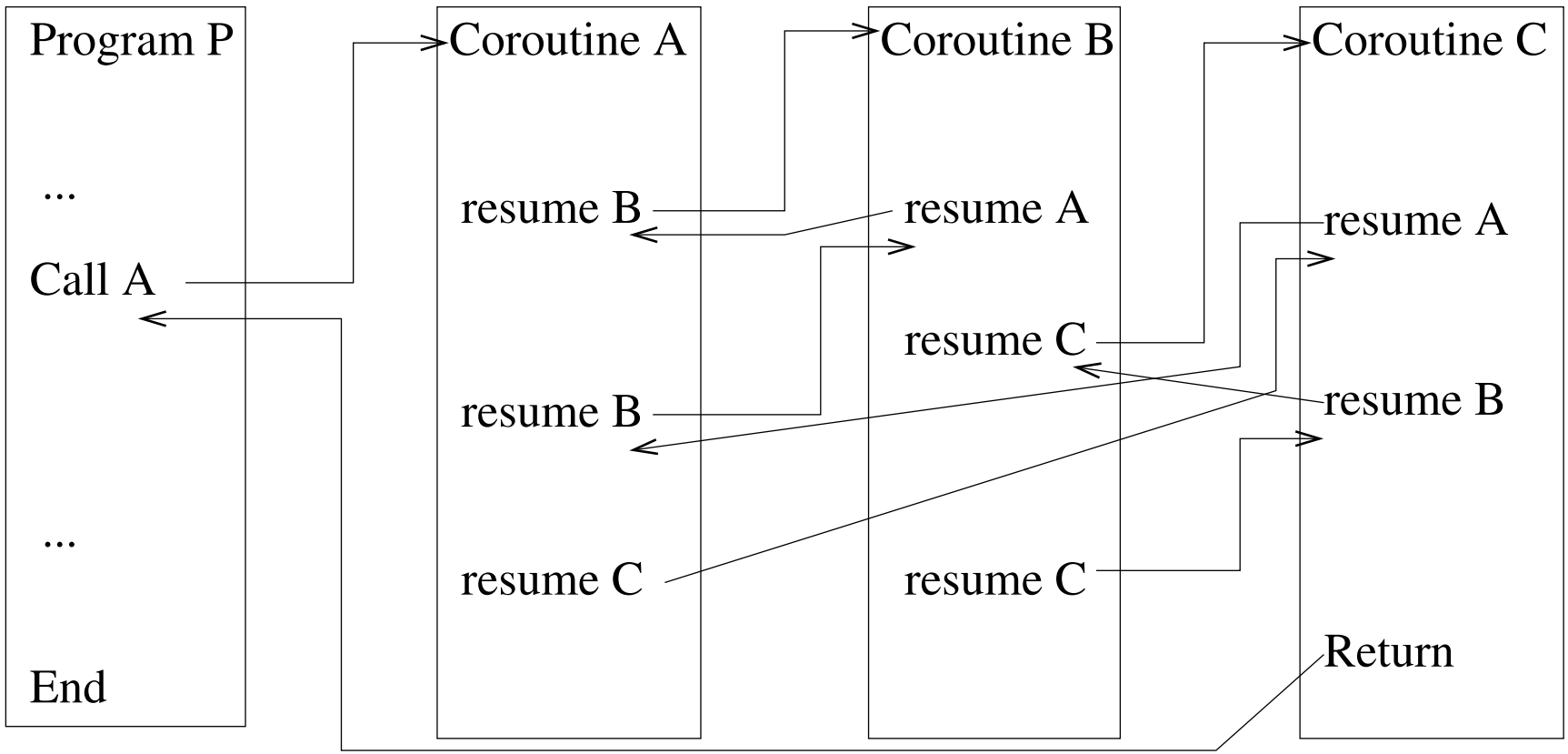
Transfert de contrôle hiérarchique



Opérations

- `resume` sauve l'état de l'exécution
- 1^{er} appel $\Rightarrow$ transfert au début de la routine
- Appels suivants $\Rightarrow$ transfert à la suite du `resume`

Opérations


Program P
...
Call A
...
End
Coroutine A
resume B
resume B
resume C
Coroutine B
resume A
resume C
resume C
Coroutine C
resume A
resume B
Return

Environnements fournissant des coroutines (ou l'équivalent)

- Boost (coroutines symmétriques et asymétriques)
- C# (fibre dans .Net 4.0 +)
- Haskell
- Javascript
- Perl
- Python
- Simula, Bliss, Modula 2

Présentation

- Méthode structurée et de haut niveau pour l'exécution concurrente
- Utilisation :

$$\texttt{cobegin } E_1\texttt{; } E_2\texttt{; } E_3\texttt{; } \ldots\texttt{; } E_n\texttt{; coend;}$$

- Chaque E_i peut contenir toutes sortes d'énoncés (y compris un cobegin)
- Autre syntaxe : $E_1 \parallel E_2 \parallel E_3 \parallel \ldots \parallel E_n$

Structure

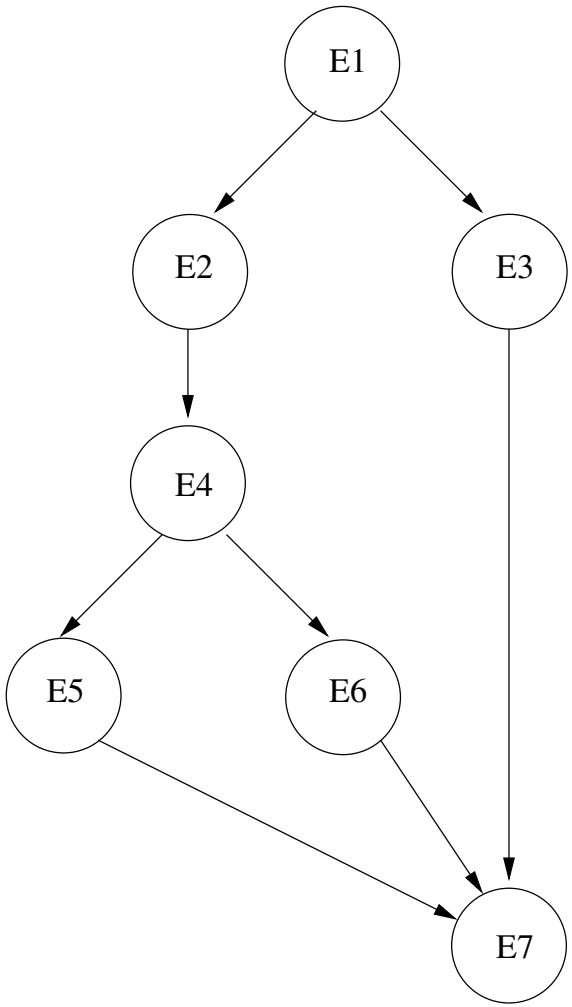
E_0; `cobegin` E_1; E_2; E_3; ...; E_n; `coend`; E_{n+1}

Graphe de précédence correspondant ? ? ? ?

Exemple

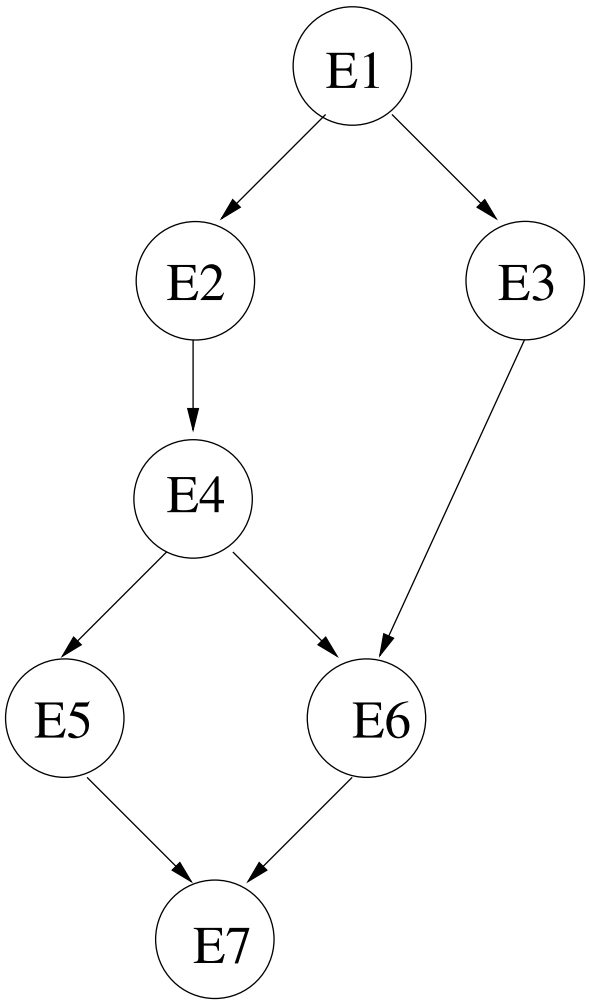
```
cobegin
    lire(a);
    lire(b);
coend
    c := a + b;
    ecrire(c);
```

Exemple




```
E1;
parbegin
   E3;
   begin
      E2;
      E4;
      parbegin
         E5;
         E6;
      parend;
   end;
parend;
E7;
```

Exemple




```
      E1;
      compte1 = 2;
      fork  L1;
      E2;
      E4;
      compte2 = 2;
      fork  L2;
      E5;
      goto  L3;
L1:   E3;
L2:   join  compte1;
      E6;
L3:   join  compte2;
      E7;
```

Exemple

```
Var   f,g : file of T;
      r,s : T;
  begin
      reset(f);
      read(f,r);
      while not eof(f)
         begin
               s := r;
               parbegin
                  write(g,s);
                  read(f,r);
               parend;
         end;
      write(g,r);
  end.
```

Concurrence explicite se fait de la façon suivante :

- Programmes $\equiv$ collection de routines séquentielles qui sont exécutées concurremment
- Déclaration : procédures
- Activation : cobegin/coend ou fork/join

Amélioration !

- Pour la lisibilité → déclaration de la routine comme un processus
- Syntaxe

```
process P1(...)
{
    ....

}
```

Avantages

- Activation implicite ou explicite
- Multiples activations

- Langages : DP, SR,JR, ADA, ...

Exemple

```
process calcul(i := 1 to n )
    var j,k:int
    write("Processus ", i)
    fa j := 1 to n ->
        c[i,j]:=0
        fa k := 1 to n ->
            c[i,j]:= c[i,j] + a[i,k]* b[k,j]
        af
     af
end
```

Au niveau de l'exécution et du partage

- Création bloquante ou non pour le parent ?
- Création permet un partage complet, partiel ou vide ?

Citoyenneté dans les langages

Approche multi-fils

VS

Approche événementielle

Citoyenneté dans les langages

- Dans les langages on parle de niveau de citoyenneté
- Niveau de citoyenneté d'un objet dépend des opérations supportées.
- On retrouve généralement les opérations suivantes possibles sur un objet :
  - être assigné à une variable
  - être passé en paramètre
  - être retourné par une fonction
  - être créé dynamiquement

Citoyenneté dans les langages

- Première classe (first class citizen)
  Supporte toutes les opérations sur les valeurs.
- Seconde classe
  Supporte juste le passage en paramètre
- Troisième classe
  Ne supporte aucune opération

Citoyenneté dans les langages

- Dans certains langages les fonctions sont des citoyens de première classe
  - elles peuvent être assignées à une variable
  - elles peuvent être passées en paramètre
  - elles peuvent être retournées par une fonction
  - elles peuvent être créées dynamiquement

Fonctions d'ordre supérieure

Une fonction est dite d'ordre supérieure si elle peut recevoir une fonction en paramètre.

Fermeture (closure)

- Dans un langage de programmation, une fermeture (closure) est un fonction accompagnée de son environnement lexical de référence
- L'environnement lexical est l'ensemble des variables non locales à la fonction qui font partie de l'environnement de référence.
- Une fermeture est un objet qui peut être passé en paramètre

Fermeture (closure)

```
def f(x):
    def g(y):  return x + y
    return g
def h(x):
    return lambda y: x + y


a = f(1)    # a est une variable contenant une fermeture
b = h(1)    # b est une variable contenant une fermeture
f(1)(5)     # fermeture "anonyme"
h(1)(5)     # fermeture "anonyme"
```

Fermeture (closure)

```
x = 1
l = [1, 2, 3]

def f(y):
    return x + y

map(f, l)
map(lambda y: x + y, l)
```

Continuation

- La continuation est une pile sémantique de ce qu'il reste à exécuter d'un programme
- Elle crée une structure de données qui représente l'état du programme à un point particulier de son exécution
- Une continuation est une forme de fermeture
- Elle est utilisée pour implanter les exceptions, les coroutines, les itérateurs, ...

Continuation

- Continuation de première classe : un langage peut sauver son état et y retourner plus tard (ne sauve pas les données)
- "Continuation passing style" : chaque fonction reçoit une fonction en paramètre qui est la continuation explicite (retour est l'appel à la fonction)
- Cela sert dans la programmation orientée événements

Continuation

```
fonction F1(valeur, F2)
{
    // Calcul du résultat
    ...

    F2(résultat)
}
```

Chapitre 2 - Concurrence et Parallélisme

Difficultés rencontrées !

Difficultés rencontrées !

1. Difficile à programmer

Difficultés rencontrées !

1. Difficile à programmer
2. Comment déterminer les activités parallèles (+)

Difficultés rencontrées !

1. Difficile à programmer
2. Comment déterminer les activités parallèles (+)
3. Mise au point (+)

Difficultés rencontrées !

1. Difficile à programmer
2. Comment déterminer les activités parallèles (+)
3. Mise au point (+)
4. Interaction

Difficultés rencontrées !

1. Difficile à programmer
2. Comment déterminer les activités parallèles (+)
3. Mise au point (+)
4. Interaction
5. Preuve de bon fonctionnement ou *model checking* (+)

Comment déterminer les activités parallèles

Caractéristiques souhaitées

- Comment déterminer que deux activités peuvent s'exécuter concurremment ?

Comment déterminer les activités parallèles

Caractéristiques souhaitées

- Comment déterminer que deux activités peuvent s'exécuter concurremment ?
- Il faut qu'elles soient disjointes.

Comment déterminer les activités parallèles

Caractéristiques souhaitées

- Comment déterminer que deux activités peuvent s'exécuter concurremment ?
- Il faut qu'elles soient disjointes.
- Qu'est-ce que des activités disjointes ?

Comment déterminer les activités parallèles

Caractéristiques souhaitées

- Soit $L(I_i) = \{a_1, a_2, ..., a_n\}$ l'ensemble de lecture de l'énoncé I_i.
- Soit $E(I_i) = \{b_1, b_2, ..., b_n\}$ l'ensemble d'écriture de l'énoncé I_i.
- Deux énoncés I_1 et I_2 sont disjoints si :
  1. $L(I_1) \cap E(I_2) = \{\}$
  2. $E(I_1) \cap L(I_2) = \{\}$
  3. $E(I_1) \cap E(I_2) = \{\}$

Mise au point

Difficulté d'appliquer des tests

- Les tests d'un programme sont basés sur l'habileté à reproduire le traitement.
- Des énoncés parallèles sont difficiles à tester car il peut être impossible de reproduire la séquence ayant produit l'erreur.

Exemple 1

```
cobegin
    x = x + 1;
    x = x + 2;
coend
```

Exemple 2 : copier un fichier

```
var  f,g : file of T;
     r,s : T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      begin
         s := r;
         parbegin
            write(g,s);
            read(f,r);
         parend
      end;
   write(g,r);
end.
```

Exemple 2 : copier un fichier

```
var  f,g : file of T;
     r,s : T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      begin
         parbegin
            s := r;
            write(g,s);
            read(f,r);
         parend
      end;
   write(g,r);
end.
```

Exemple 2 : copier un fichier

```
var  f,g : file of T;
     r,s : T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      begin
         parbegin
            a : s := r;
            b : write(g,s);
            c : read(f,r);
         parend
      end;
   write(g,r);
end.
```

Preuve et *model checking* ! ! !

Preuve et *model checking* ! ! !

1. De plus en plus utilisée avec les méthodes formelles

Preuve et *model checking* ! ! !

1. De plus en plus utilisée avec les méthodes formelles
2. Propriétés à vérifier :

Preuve et *model checking* ! ! !

1. De plus en plus utilisée avec les méthodes formelles
2. Propriétés à vérifier :
   - Sûreté
     Aucun évènement indésirable ne doit se produire

Preuve et *model checking* ! ! !

1. De plus en plus utilisée avec les méthodes formelles
2. Propriétés à vérifier :
   - Sûreté
     Aucun évènement indésirable ne doit se produire
   - Vivacité
     Des évènements souhaitables doivent se produire

Preuve et *model checking* ! ! !

1. De plus en plus utilisée avec les méthodes formelles
2. Propriétés à vérifier :
   - Sûreté
     Aucun évènement indésirable ne doit se produire
   - Vivacité
     Des évènements souhaitables doivent se produire
   - Équité