Une introduction à l'algorithmique du parallélisme avec occam-π

Gilbert Dhuime

Introduction

The π-calculus differs from other models of communicating behaviour mainly in its treatment of mobility. The movement of a piece of data inside a computer program is treated exactly the same as the transfer of a message- or indeed an entire computer program- across the internet .One can also describe networks which reconfigure themselves.

Robin Milner : Communicating and Mobile Systems : the π-calculus

L'évolution récente des processeurs multi-coeurs, l'extension extraordinaire d'internet, notamment dans le domaine de la téléphonie mobile, imposent une vision renouvelée de la gestion du parallélisme et de la communication entre processus concurrents.
Pour ce qui concerne les processeurs, la finesse de gravure des puces a pratiquement atteint ses limites et l'évolution multicoeurs impose l'exécution en paralléle des processeurs implantés sur une même puce.

Les machines dédiées au calcul intensif utilisent plus de dix mille puces multicoeurs interconnectées par fibres optiques. Dans un futur proche ces connections seront optoélectroniques.
Les grands fondeurs comme IBM développent des prototypes de macro-composants constitués de trois couches qui intègrent une grappe de processeurs multicoeurs (premiére couche), de la mémoire (deuxième couche) et de l'optoélectronique (troisième couche) qui assurerait un débit de 70 Terabits/sec entre les éléments constitutifs d'un macro-composant (processeurs et mémoire) et un débits de 1 Terabits/sec entre macro-composants.

Le site d'IBM : `http:// www.research.ibm.com/photonics`
expose l'état de l'art sur ce sujet.

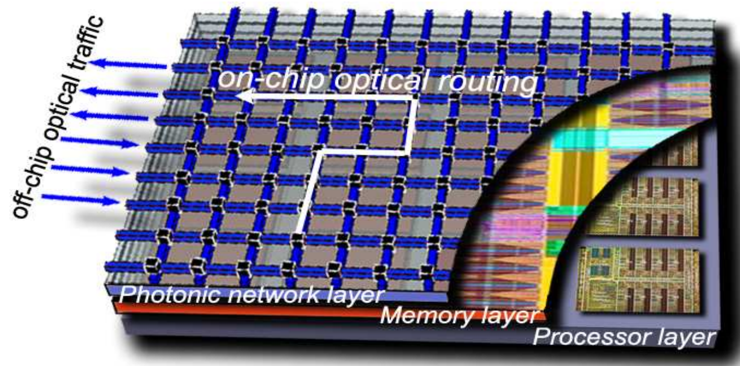


FIGURE 1 – Un macro composant IBM

[1] Au début des années 1980 la société INMOS proposait un langage, Occam, et un microprocesseur dédié : le Transputer. Ce dernier, doté de quatre canaux d'entrées sorties, était conçu pour s'intégrer en standart à un réseau de processeurs.

Le tandem Occam-Transputer apportait une réponse à la gestion, tant sur le plan du matériel que du logiciel, d'un parallélisme à grain fin dans le cadre d'architectures distribuées.
Language de programmation de haut niveau bien conçu pour traduire les algorithmes distribués sur des architectures les plus variées Occam , basé sur les travaux théoriques de Tony Hoare [Hoa04] [Ros98]et de ses collaborateurs , donnait naissance à un nouveau style de programmation dû à David May chez INMOS résolumment **orienté processus** (en occam une simple affectation **est** un processus). Malheureusement ces idées, trop en avance, ne trouvèrent pas l'accueil escompté et INMOS fut racheté par THOMSON-SGS. Occam, arrivé en pleine maturité dans sa version 2.1 [Tho95] connut un sursis d'estime mais, finalement, le projet autour du couple Occam-Transputer dù être abandonné.
Au moment de son abandon, Occam proposait un style de programmation concurrente statique où tous les acteurs du langage (processus, ressources diverses, canaux de communication, etc ..) devaient être entièrement déterminés à la compilation.
Entre temps les travaux de Robin Milner, [Mil07], [MPW92] , Joachim Parrow et David Walker sur le π-calculus montraient le chemin conduisant à une extension d'Occam en occam-π où la mobilité prenait toute sa place .
La mobilité des données , celle des canaux qui permet à un réseau de se reconfigurer et finalement la mobilité des processus dans les réseaux confèrent à occam-π une place de choix dans l'évolution que nous avons évoquée que connait actuellement l'informatique.
Ce travail d'extension du langage d'Occam2.1 en occam-π est dû principalement aux chercheurs de l'Université de Kent qui ont mis au point le compilateur

1. Guillaume d'Occam était un moine anglais franciscain du XIV ème siècle connu pour avoir énoncé un principe dit du "rasoir d'Occam". Ce principe semble être un principe d'économie qui veut que dans toute théorie il faille utiliser le plus petit nombre possible d'hypothèses et ainsi élaguer tout ce qui est superflu.

Kroc (Kent retargetable occam compiler) qui sera pour nous le compilateur de référence. L'avantage qu'un langage comme Occam présente par rapport à ceux actuellement utilisés dans le domaine du parallélisme est qu'il est fondé sur un modèle mathématique éprouvé de la concurrence (CSP) ce qui permet de valider les projets basés sur ce formalisme. FDR est un de ces outils de validation.
Les renseignements relatifs à l'implémentation fdr2 de FDR peuvent être trouvés à l'adresse de Formal Systems : `http://www.fsel.com/`.
Occam permet de se former très rapidement à l'algorithmique du parallélisme sur des architectures distribuées. Il se prête bien au prototypage dans les domaines les plus variés et à la robotique.

Dans la présentation du langage, de sa syntaxe notamment, je me suis principalement inspirés du manuel de référence original d'INMOS sur Occam2.1 paru chez Prentice Hall et que l'on peut librement télécharger à :
`http://www.wotug.org/occam/documentation/oc21refman.pdf` .
Un exposé récent sur la syntaxe d'occam-π dù à Peter Welch peut être téléchargée à :
`https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference`.

Ce livre, issu de mon expérience d'enseignant, se veut avant toute chose pratique. Dans un domaine encore relativement neuf l'important est que le lecteur puisse "mettre le pied à l'étrier " le plus vite possible sans perdre de temps à des problèmes liés à la mise en oeuvre des outils de développement et à la recherche de documentation. Dans ce but on trouvera de nombreuses références au language ainsi qu'aux publications qui lui sont associées sur internet. Je me suis efforçé d'illustrer par de nombreux exemples, le plus souvent brefs, les concepts liés à occam-π au fur et à mesure de leur présentation.

Il se compose de quatre parties :

La première partie comprend les chapitres 1 à 13 qui couvrent occam dans sa version "classique" qu'est occam2.1 avec les modifications syntaxiques mineures dues à l'actuelle implémentation. L'étude de ces chapitres est essentielle pour la compréhension des chapitres suivants .

La seconde constituée des chapitres 14 à 17 est essentiellement consacrée à l'-exposition des notions issues du π-calculus : mobilité des données , des canaux, des processus et des barrières.

La troisième présente quelques applications basées sur les librairies de Kroc comme celles consacrées à la programmation réseau et au graphisme. Il nous a paru également important d'y faire un exposé sur CIF qui est l'interface entre occam-π et C.

Dans une derniére partie sont exposées des applications sur les différents modes de communication dans les réseaux décrits par des graphes d'interconnection classiques comme les grilles, les hypercubes, les arbres, de même que les algorithmes de routage qui leur sont associés ainsi que quelques algorithmes qui nous semblent représentatifs d'une classe constituée de processus très élémentaires mais qui, dans une architecture appropriée, effectuent une tache non triviale.

Au chapitre 21 intitulé 'Les outils' on trouvera les renseignements nécessaires à l'installation de Kroc sous Linux (Ubuntu), de même que celle de l'éditeur repliable Kate.

Table des matières

1 Syntaxe et format des programmes **1**

1.1 Notations syntaxiques 1

1.1.1 Les commentaires 4

1.1.2 Identificateurs et mots clés 4

1.2 Structure d'un programme occam 5

2 Les types **9**

2.1 Les types associés aux données 9

2.1.1 Les types primitifs 10

2.1.2 Les types structurés 10

Le type data.table 10

Le type data.struct 11

2.1.3 Renommer un type de données 11

2.2 Les types associés aux canaux 12

2.2.1 Les protocoles simples 12

2.2.2 Les protocoles séquentiels 12

2.2.3 Les protocoles variables 13

2.2.4 Déclarer un protocole 13

2.3 Le type timer 14

2.4 Le type barrière 14

2.5 Les types mobiles 14

2.5.1 Généralités sur les types mobiles 14

3 Les variables et les constantes 17
3.1 Les déclaration de variables 17
3.1.1 Déclaration de variables sans initialisation 17
3.1.2 Déclaration de variables avec initialisation 18
3.1.3 Les éléments de tableau 19
3.1.4 Les segments de tableaux 20
3.1.5 Les éléments de structure 22
3.2 Les déclarations de constantes 23
3.2.1 Les littéraux 23
3.2.2 Les constantes 24

4 Les expressions 27
4.0.3 Les opérateurs arithmétiques 28
4.0.4 Les opérateurs sur les bits 28
4.0.5 Les opérateurs de décalage 28
4.0.6 Les opérateurs arithmétiques modulo 29
4.0.7 Les opérateurs booléens 30
4.0.8 Les opérateurs relationnels 30
4.0.9 Quelques opérateurs spécifiques 30
MOSTNEG et MOSTPOS 30
L'opérateur SIZE 31
L'opérateur AFTER 31
4.0.10 Les opérateurs de conversion 31
Conversions entre entiers 31
4.0.11 Conversion entiers , flottants 32
4.0.12 Conversion flottant , flottant 33
4.0.13 Exercice de récapitulation 34

5 Les processus primitifs 39
5.1 Affectation 40
5.2 Communication 40
5.2.1 Déclarer un canal 41
5.2.2 La communication 42
La lecture d'un canal 42
L'écriture d'un canal 43

5.2.3 Exemples complets de lectures et d'écritures 44
protocoles simples 44
protocoles sequentiels 47
protocoles variables 47
5.3 Le timer 48
5.3.1 Déclarer un timer 49
5.3.2 Lecture d'un timer 49
5.4 La synchronisation sur une barrière 50
5.4.1 Déclarer une barriere 50
5.4.2 Synchronisation 50
5.5 SKIP et STOP 52

6 Les processus séquentiels 53
6.1 Les processus SEQ 53
6.1.1 Les processus SEQ répliqués 54
6.2 Les processus IF 55
6.2.1 Les processus IF répliqués 56
6.3 Les processus CASE 57
6.4 Les processus WHILE 58

7 Les processus parallèles 61
7.1 Les processus PAR 61
7.1.1 Les processus PAR répliqués 64
7.1.2 Circulation sur un anneau 64

8 Les processus alternatifs 67
8.1 Les processus ALT 67
8.1.1 Les processus alternatifs répliqués 69
8.1.2 Multiplexage 70
8.1.3 Processus ALT prioritaire 72
Implémanter un chien de garde (watchdog) 72
8.2 La gestion des évènements asynchrones 74

9 Les processus nommés et les fonctions **77**
9.1 Les processus nommés 77
9.1.1 La déclaration d'un processus nommé 80
9.1.2 Référencer un processus nommé 81
9.1.3 Un programme de trie simple 82
9.1.4 Les cinq philosophes revisités (alting) 84
9.2 Les processus nommés récursifs 90
9.3 Les fonctions 92
9.3.1 Référencer une fonction 94

10 Portée des déclarations **95**
10.1 Les déclarations générales 95
10.2 Les déclarations de ressources 95
10.2.1 Le cas particulier des processus parallèles 97

11 Les canaux partagés en occam-π **99**
11.1 Les blocs CLAIM 100
11.2 Partage entre plusieurs écrivains et un lecteur 100
11.3 Partage entre plusieurs lecteurs et un écrivain 102
11.4 Partage entre plusieurs écrivains et plusieurs lecteurs 104
11.4.1 Les cinq philosophes revisités (shared) 106

12 La mobilité des données en occam-π **111**
12.1 Le type mobile.data.type 111
12.2 La déclaration des variables mobiles 111
12.2.1 La création des variables mobiles 112
12.2.2 Perte de référence, clonage 112
12.3 Une barrière partielle 114

13 La mobilité des canaux en occam-π **117**
13.1 La déclaration CHAN TYPE 117
13.1.1 La déclaration d'un canal mobile 118
13.1.2 La création d'un canal mobile 118
13.1.3 Les canaux constitutifs d'un canal mobile 119
13.2 La mise en oeuvre de la mobilité 120
13.2.1 Passage des bouts par référence 120
13.2.2 Passage des bouts par lecture d'un canal 121
13.2.3 Un serveur simple 123
13.2.4 Lectures et écritures multiples entre processus 125

14 La mobilité des processus en occam-π 131

14.1 La déclaration PROC TYPE 132
14.2 Les processus mobiles 132
14.3 La suspension d'un processus mobile 133
14.4 La mobilité d'un processus dans un réseau 136
14.4.1 Un exercice de synthèse 138

15 Programmation système 141

15.1 Les fichiers 141
15.2 La programmation réseaux 142
15.2.1 Les sockets 142
15.3 Le forking 146
15.3.1 Communication avec les canaux mobiles 148
15.3.2 Synchronisation avec les barrières mobiles 152
15.3.3 Un serveur multi tâches 153
15.4 CIF et l'interface avec le language C 156
15.4.1 Un premier exemple 157
15.4.2 Autre version du premier exemple 159
15.4.3 Parallélisme (1) 159
15.4.4 Parallélisme (2) 161
15.4.5 Les canaux mobiles 162

16 Programmation graphique 165

16.1 Introduction 165
16.2 La courbe de Hilbert 167
16.3 Le problème des deux corps 171
16.3.1 Mise sur orbite d'un satellite 177

17 La grille 181

17.1 La grille torique 181
17.1.1 Le routage store and forward 182
17.1.2 Le programme 183
17.1.3 La structure d'un processus 183
17.1.4 La structure du routeur 184
17.1.5 Le programme complet 185

17.1.6 Utiliser le parallélisme des liens 188
17.2 Un programme de trie sur la grille 191
17.2.1 Le principe de l'algorithme 191
17.2.2 Mise en oeuvre de l'algorithme 193
17.3 Le routage whorm hole 202
17.3.1 Implantation des canaux virtuels 202
17.3.2 Le protocole adopté 203
17.3.3 Le routage 204
L'unité centrale 204
Le routeur 204
Le programme 205

18 L'hypercube **209**
18.1 Généralités 209
18.2 Le codage des canaux 210
18.3 Le routage 211
18.4 Utiliser le parallélisme des liens 215
18.4.1 Routage à distance $k \leq n$ dans $H(n)$ 215
18.4.2 Cas général 219
18.5 Diffusion 223

19 Le graphe complet **227**
19.1 Le codage des canaux mobiles 228
19.2 Le processus init.net 228
19.2.1 Phase un de init.net() 229
19.2.2 Phase deux de init.net() 229
19.3 Les processus process() 230
19.3.1 La matrice c.chan.link 230
19.3.2 Phase un de process() 231
19.3.3 L'échange total 231

20 Arithmétique **235**
20.1 L'algorithme d'addition à retenue anticipée 235
20.2 Mise en oeuvre de l'algorithme 239
20.2.1 le codage des canaux 239
20.2.2 Le programme de test 240

21 Les outils **243**

21.1 Le compilateur Kroc 243

21.1.1 Les prérequisites 243

21.1.2 Installer kroc 243

21.1.3 Faire reconnaitre les librairies de kroc 244

21.1.4 La compilation séparée 244

21.1.5 Ecrire ses propres librairies 245

21.2 L'éditeur Kate 246

22 Les modules de kroc **249**

Chapitre 1

Syntaxe et format des programmes

1.1 Notations syntaxiques

La description de la syntaxe d'un programme Occam, très proche de la notation Backus Naur , s'en écarte néanmoins en incorporant l'indentation comme partie intégrante de ses constructions.
L'exemple ci dessous qui définit la syntaxe d'un processus séquentiel illustre ce fait.

```
proc.sequentiel  => SEQ
                      {processus}
```

Qui se lit : Un processus séquentiel se déclare par le mot clé SEQ sur une seule ligne. A la ligne suivante, indenté de deux espaces à compter du S de SEQ sont écrites les déclarations éventuellement vide de processus. Toutes les déclarations des processus (s'il y en a au moins une) constitutifs d'un processus séquentiel introduit par SEQ sont **alignées**.
Par exemple :

```
SEQ
  processus.1
  processus.2
  processus.3
```

De la même façon un processus parallèle se déclare par :

```
proc.parallel  => PAR
                    {processus}
```

Par exemple :

```
   PAR
     INT x, y : -- variables locales au processus SEQ qui suit
     SEQ
       -- en Occam une affectation est un processus
       x:= 6
       y:= 2*x
     INT v, w : -- variables locales au processus SEQ qui suit
     SEQ
       v:= 5*w
       w:= 20*t
```

Cet exemple déclare un processus parallèle composé de deux processus séquentiels, eux mèmes composés de deux affectations qui, en Occam, sont des processus.

L'indentation doit être respectée partout où elle apparait dans la syntaxe.
Par exemple :

```
proc.if                  => IF {?,replicateur}
                              {choix}
choix                    => expression
                              processus
```

Dans l'écriture de 'choix' le texte de 'processus' doit être indenté de deux caractères par rapport au premier caractère de 'expression'.

Exemple :

```
IF
   a = b   --   expression
     SKIP  --   processus SKIP
   a <> b  --   expression
     STOP  --   processus STOP
```

{ item } fait reférence à une liste, éventuellement vide, d'items tous au même niveau d'indentation à raison d'un item par ligne.

{ 0, liste } fait reférence à une liste éventuellement vide dont les éléments sont écrits sur une même ligne. Si cette liste est non vide ses éléments sont séparés par une virgule.
L'exemple ci dessous définit la syntaxe de la déclaration d'un processus nommé. La liste des paramétres qui suit l'identificateur du processus est une liste éventuellement vide.

```
processus.nomme      => PROC process.id ({0, parametres.formels})
                          {general}    -- declarations structurelles
                          {ressource}  -- declarations de ressources
                           processus
                        :
process.id           => m.identificateur
```

{ 1, liste } fait reférence à une liste d'au moins un élément tous écrits sur une même ligne. S'il y a plus d'un élément ceux ci sont séparés par une virgule.
Par exemple reprenant la définition de l'affectation vue plus loin au chapitre 5 on définit l'affectation par :

```
affectation          => {1,variable.id} := {1,expression}
```

Exemples :

```
var := TRUE
a,b,c := 1,2,3
```

Qui se lit : une affectation est définie par une liste d'au moins une variable, suivie de ' :=', suivie d'une liste d'au moins une expression (les deux listes de par et d'autre de ' :=' doivent avoir le même nombre d'éléments).

{ ?,construction.syntaxique } fait référence à une construction syntaxique optionnelle.
L'exemple qui suit est assoçié à la définition d'un processus alternatif qui peut être non seulement prioritaire (?,PRI) mais aussi répliqué (?,replicateur).

```
alternatif  => {?,PRI} ALT {?,replicateur}
                     {alternative}
```

Le caractère '|' marque un choix disjonctif. Si plusieurs choix ne peuvent pas tenir sur une même ligne ils sont repris à la ligne suivante derrière '=>' et alignés sur la précédente écriture.
Par exemple :
Les différents types de processus en Occam sont définis par :

```
processus     => proc.primitif | proc.standart
              => proc.instance | fonc.instance
              => proc.mobile   | proc.cif
```

En principe un programme Occam possède une déclaration par ligne et l'indentation des déclarations au sein d'un même programme est partie inhérente de la syntaxe. Cette indentation est toujours de deux caractères blancs. Ce fait doit être pris en considération lors du choix d'un éditeur de texte et des facilités qu'il propose pour le paramétrer de façon à être compatible avec les exigences d'Occam.

Continuation d'une ligne
Comme il vient d'être dit une déclaration doit pouvoir tenir sur une seule ligne.
Il y a des situations où ce n'est pas possible .
Ces situations sont restreintes aux contraintes suivante :
La césure peut se faire :
après un opérateur : +, - , * , /
après une virgule , un point virgule, l'opérateur d'affectation := , les mots clés IS, FROM et FOR.

1.1.1 Les commentaires

Un commentaire est introduit par deux tirets - - et il s'étend jusqu'à la fin de la ligne.
Un commentaire **ne peut pas être indenté moins** que la déclaration qui le suit mais par contre peut être plus indenté que cette déclaration. Par exemple :

```
  --  ce SEQ declare un premier processus sequentiel
      --  le commentaire ci dessus est correct
      --  celui ci egalement
  SEQ
    processus
    processus
    ..
    processus

--  SEQ declare un deuxieme processus sequentiel
--  le commentaire ci dessus est incorrect
--  celui ci egalement
    SEQ
      processus
      processus
      ..
      processus
```

Une bonne pratique est de mettre les commentaires au même niveau d'indentation que la déclaration qui leur succède.

1.1.2 Identificateurs et mots clés

Tout identificateur commence par un caractère alphabétique, suivi de caractères alphanumèriques où de points .Tous les mots clés d'Occam comme SEQ, PAR , sont uniquement constitués de majuscules.
Attention les identificateurs sont sensibles à la casse des caractères. Ma.variable est un identificateur distinct de ma.variable.
Exemples d'identificateurs valides :
Transputer, Mavariable, vt200, cannal6.in ,mon.lien.v6, LinkOut etc ..

Conventions recommandées en occam-π

En occam-π on recommande que les identificateurs ne comportent que des caractères alphanumériques séparés éventuellement par des points. Le premier caractère est obligatoirement un caractère alphabétique.
Nous suivrons cette directive dans tout ce qui suit.
M.identificateur désigne un identificateur composé uniquement de majuscules , de chiffres où, de points qui commence par une majuscule
m.identificateur désigne un identificateur composé uniquement de minuscules , de chiffres où, de points qui commence par une minuscule.

```
identificateur => M.identificateur
               => m.identificateur
```

Par exemple les identificateurs associés aux mots clé du language comme PROC,SEQ, PAR sont des M.identificateurs.
Les identificateurs de canaux, lorsqu'ils sont utilisés en tant que paramétres, doivent se terminer par les caractères ? où ! suivant qu'ils sont définis en lecture (resp en écriture).

Exemple : Dans la déclaration qui suit kbd ? désigne un canal de protocole BYTE utilisé en lecture et scr !, err ! deux canaux de protocole BYTE utilisés en écriture.

```
 PROC main(CHAN BYTE kbd?, scr!, err!)
```

1.2 Structure d'un programme occam

Un programme Occam est constitué de déclarations **toutes au même niveau zéro d'indentation**.
Ces déclarations peuvent être précédées de directives qui seront prises en charges par le compilateur (include, pragma) et (ou) par l'éditeur de liens (use).

Les déclarations sont visibles de toutes celles qui leur succèdent dans l'ordre de l'écriture.
En particulier les déclarations de protocole, de constantes, de structure qui précèdent les déclarations de processus nommés sont visibles dans tous les textes de ces processus.

```
program.occam        => {directives}
                        {general}


directives           => include | use | pragma
include              => #INCLUDE "nom.de.fichier"
use                  => #USE "nom.de.fichier"
```

```
pragma               => #PRAGMA text.pragma

general              => data.struct | renommage
                     => protocole
                     => constante
                     => process.nomme | fonction
                     => mobil.type
```

Exemple :

L'exemple ci dessous donne le squelette typique d'un programme en occam.
Ce programme déclare le protocole STR qui décrit la nature des informations circulant dans le canal ptt.canal, deux processus nommés client() et serveur() et le processus nommé main().

```
PROTOCOL STR IS INT::[]BYTE:

PROC serveur(CHAN STR f.canal!)
  --
  -- texte du processus serveur
:

PROC client(CHAN STR f.canal?, CHAN BYTE ecran!)
  --
  -- texte du processus client
:

PROC main(CHAN BYTE scr!)
  CHAN STR ptt.canal  :
  PAR
    serveur(ptt.canal!)
    client(ptt.canal?, scr!)
:
```

L'écriture d'un programme Occam doit déclarer au moins un processus nommé et la dernière déclaration doit être celle d'un processus nommé.
L'identificateur de ce processus importe peu mais, c'est le choix de l'auteur, pour rester fidèle à des conventions issues du monde C ou Java nous l'appelerons **main**.
Le processus nommé main() est celui qui est lançé en premier lorsque le programme est exécuté.
On remarque que sont associés à main(), en tant que paramètres, des canaux d'entrées sorties permettant les lectures clavier, les écritures à l'écran et l'affichage des erreurs. Ces canaux peuvent être présents en tout où en partie. Dans l'exemple qui suit seul le canal scr ! d'écriture à l'écran est utilisé.

Exemple :

```
--ch1_1.occ
--Mon premier programme en Occam avec Kroc
--La processus out.string() appartient a
--la librairie course.lib de Kroc et permet
--d'afficher une chaine de caracteres a l'ecran
--en utilisant le canal scr

#USE "course.lib"

PROC main(CHAN BYTE scr!)
  SEQ
    out.string("Hello Occam ...*n",0,scr!)
    out.string("BY ... *n",0,scr!)
:
```

Si le texte de ce programme est dans un fichier **premier.occ** et si, dans un terminal, on se place dans le répertoire contenant ce fichier alors il sera compilé et linké à la librairie **course** par :
kroc premier.occ -lcourse.
Un exécutable **premier** est alors crée dans le même répertoire que le texte source et peut être lançé par la commande : **./premier.**

Chapitre 2

Les types

Un type caractérise une ressource et permet de vérifier, à la compilation, la cohérence liée à son emploi. Les processus en s'exécutant utilisent, échangent et, éventuellement, modifient des ressources.
A ce titre les canaux, les timers et les barrières sont des ressources. En occam-π la notion de ressource s'étend à la mobilité des données, des canaux et des processus.

Les types data.type sont associés aux données de nature numériques ou logiques.
Les types channel.type sont associés à des lectures (resp écritures) d'un canal.
Les types timer.type sont associés à la lecture du temps ou d'un délai dans le temps.
Les types barrière.type sont associés à des synchronisations entre processus.
Les types mobile.type sont associés à des ressources crées dynamiquement et transférables dans un réseau : des données mobiles, des canaux mobiles, des barrieres mobiles, des processus mobiles.

```
type            => data.type
                => channel.type
                => timer.type
                => barriere.type
                => mobile.type
```

2.1 Les types associés aux données

Les types de données primitifs reconnus par occam-π sont les entiers, les flottants et les booléens.
A partir de ces types primitifs sont construits les types plus élaborés que sont les types data.table et les types data.struct. Par ailleurs un type peut être renommé.

```
data.type            => primitive.type | structured.data.type
                     => DATA TYPE M.identificateur IS data.type :

structured.data.type => data.table  | data.struct
```

2.1.1 Les types primitifs

Les types de données primitifs supportés par Occam sont :

```
primitive.type    => BOOL  | BYTE  | INT    | INT16
                  => INT32 | INT64 | REAL32 | REAL64
```

BOOL . Booléens dont les valeurs sont TRUE et FALSE.

BYTE . Entiers **non signés** codés sur 8 bits .
La valeur d'un BYTE varie de 0 à 255 (de #00 à #FF en hexadécimal).

INT . Entiers signés repréentés en complément à deux .Leur codage n'est pas précisé et dépend de l'implémentation . On peut cependant le déterminer indirectement en examinant les valeurs produites par les opérateurs MOSTPOS (resp MOSTNEG) qui, appliqués à un type entier donnent la plus grande valeur positive (resp la plus petite valeur négative) associée à ce type. Par exemple si MOSTPOS INT vaut $(2^{31} - 1)$ alors on en conclue que INT est implémenté sur 32 bits **mais** INT32 bien que dénotant le même codage doit être considéré comme étant de type différent !.

Le type INT est pris par défaut comme le type des constructions répliquées associées à SEQ, PAR, ALT ainsi qu'aux dimensions des data.tablex et aux opérateurs de décalages.

INT16 : Entiers signés en complément à deux codés sur 16 bits .
Leur valeur varie de -2^{15} à $(2^{15} - 1)$ soit de -32768 à 32767.

INT32 : Entiers signés en complément à deux codés sur 32 bits .
Leur valeur varie de -2^{31} à $(2^{31} - 1)$.

INT64 : Entiers signés en complément à deux codés sur 64 bits .
Leur valeur varie de -2^{63} à $(2^{63} - 1)$.

REAL32 : Flottants codés sur 32 bits comprenant 1 bit de signe , 8 bits d'exposant et 23 bits de mantisse en conformité avec le standart ANSI/IEEE 754-1985 .

REAL64 : Flottants codés sur 64 bits comprenant 1 bit de signe , 11 bits d'exposant et 52 bits de mantisse en conformité avec le standart ANSI/IEEE 754-1985.

2.1.2 Les types structurés

Le type data.table

Le type data.table définit une zone mémoire à laquelle on peut accéder dans sa totalité où à un sous ensemble contigü d'éléments appelé segment.
Si ce sous ensemble est à 1 élément on retrouve la notion d'élément de tableau classique.
Syntaxe :

```
data.table => [?, expression] data.type
```

L'expression entre crochets doit être de type INT. Dans le cas de déclarations de constantes elle peut être omise car elle est déterminée implicitement à la compilation (cf : le paragraphe 3.2.2 du chapitre 3).

Exemples :

```
[40] BOOL   --
[10][5]INT  --
[20]REAL64  --
--
[]BYTE      -- type de la constante message definie ci dessous
VAL []BYTE message IS "hello world !" :
```

Le type data.struct

Un data.table caractérise une zone de mémoire dont les élêments ultimes que l'on peut en extraire sont de même type. Un type.struct caractérise une zone mémoire dont les éléments ultimes qui peuvent en être extraits, les champs, sont de types différents.

```
data.struct      => DATA TYPE struct.type.id
                      record
                    :
record           => RECORD
                      {type {1, champ}:}
champ            => m.identificateur
struct.type.id   => M.identificateur
```

Exemple :

```
DATA TYPE MY.STRUCT
  RECORD
    BYTE b0 , b1     :
    [10]BYTE  str    :
:
```

Dans cet exemple MY.STRUCT est l'identificateur d'un type data.struct à trois champs. Les éléments ultimes qui peuvent être extraits d'une variable de ce type sont de type BYTE et de type [10]BYTE.

2.1.3 Renommer un type de données

Il peut être parfois utile, à des fins de lisibilité, de renommer un type de données d'autant que ce renommage est pris en charge par le compilateur.
Par exemple les écritures suivantes définissent trois nouveaux types : LONGUEUR, AIRE, STRING.
Bien que LONGUEUR et REAL32 codent les mêmes informations ils sont considérés comme deux types différents par le compilateur.

```
DATA TYPE LONGUEUR IS REAL32  :
DATA TYPE AIRE IS REAL32      :
DATA TYPE STRING IS [255]BYTE :
```

2.2 Les types associés aux canaux

Les protocoles décrivent la structure de l'information qui transite dans les canaux et déterminent le type de ces derniers.

```
type.primitif       => channel.type

channel.type        => CHAN protocole.id
protocole.id        => M.identificateur
```

La conformité des données lues où écrites selon le protocole associé à un canal est vérifiée à la compilation.
On peut, par soucis de lisibilité, déclarer un protocole. UNe déclaration est obligatoire pour les protocoles variables.

```
protocole           => protocole.fixe
                    => protocole.variable
                    => PROTOCOL protocole.id IS protocole : -- declaration de protocole

protocole.fixe      => protocole.simple
                    => protocole.sequentiel
```

2.2.1 Les protocoles simples

```
protocole.simple  => data.type | type.primitif::[]data.type
```

Exemples :

```
On suppose la structure MY.STRUCT declaree .

CHAN INT16         -- 1 : canal de protocole  INT16
CHAN [10]REAL32    -- 2 : canal de protocoles [10]REAL32
CHAN MY.STRUCT     -- 3 : canal de protocole  MY.STRUCT
[20]CHAN [5]INT    -- 4 : tableau de 20 canaux , chaque canal est de protocole [5]INT

CHAN INT:: []BYTE  -- 5 : canal de protocole INT::[]BYTE
```

–1 définit un canal qui ne véhicule que des variables de type INT16.
–2 définit un canal qui ne véhicule que des variables de type de [10] REAL32.
–3 définit un canal qui ne véhicule que des variables de type MY.STRUCT.
–4 définit un tableau de 20 canaux . Chacun de ces canaux véhicule une variable de type [10]REAL32.
–5 définit un canal qui ne véhicule que des blocs de données constitués d'un INT suivi d'un tableau de BYTE.

2.2.2 Les protocoles séquentiels

Un protocole séquentiel est défini comme une liste non vide dont les éléments sont des protocoles simples séparés par un point virgule.
Pour des raisons de lisibilité il est recommandé de déclarer les protocoles séquentiels.

```
protocole.seq        => {1; protocole.simple}
```

Exemples :

```
PROTOCOL COMPLEX IS REAL32 ; REAL32  : -- COMPLEX denote le protocole sequentiel REAL32 ; REAL32
PROTOCOL PROTSEQ IS REAL32 ; INT ; INT::[]BYTE :

CHAN COMPLEX      -- canal de protocole COMPLEX
CHAN PROTSEQ      -- canal de protocole PROTSEQ
```

2.2.3 Les protocoles variables

Un protocole variable permet à un canal de véhiculer des informations de protocoles différents. Un code préalable, le tag, permet de discriminer entre ces différents protocoles. Ces derniers doivent être des protocoles séquentiels.
Un protocole variable doit faire l'objet d'une déclaration .

Syntaxe :

```
protocole.variable   => PROTOCOLE protocole.id
                          CASE
                            {tagged.protocol}
                        :

tagged.protocol      => tag | tag ; protocole.sequentiel
tag                  => m.identificateur
protocole.id         => M.identificateur
```

Exemple :

```
PROTOCOL PRTV
  CASE
    char    ; BYTE
    complex ; REAL32 ; REAL32
    string  ; INT::[]BYTE
    halt
:
--
--
CHAN PRTV   -- canal de protocole PRTV. Il peut vehiculer un BYTE ou deux REAL32
            -- ou un INT suivi d'un tableau [] BYTE, ou ..  rien du tout
            -- suivant la valeur  du tag : char, complex, string, halt.
```

2.2.4 Déclarer un protocole

Déclarer un protocole est une bonne pratique dès lors que le protocole n'est pas associé à un type primitif (BYTE, INT , REAL32 , etc ...). Syntaxe :

```
protocole    => PROTOCOL protocole.id IS protocole :
```

Exemples :

```
PROTOCOL COMPLEX IS REAL32 ; REAL32 :
PROTOCOL STRING   IS INT::[]BYTE    :
--
--
CHAN COMPLEX    -- canal de protocole COMPLEX
CHAN STRING     -- canal de protocole STRING
```

2.3 Le type timer

Le timer est associé au temps. Son étude fait l'objet du paragraphe 5.3 du chapitre 5.

Syntaxe :

```
timer.type       => TIMER
```

2.4 Le type barrière

Une barrière permet à des processus concurrents de se synchroniser. L'étude des barrières est faite au paragraphe 5.4 du chapitre 5.

Syntaxe :

```
barriere.type   => BARRIER
```

2.5 Les types mobiles

Les types mobiles sont associés à des ressources crées dynamiquement et transférables dans un réseau.

```
mobile.type     => mobile.data.type
                => mobile.barrier.type
                => mobile.channel.type
                => mobile.process.type
```

2.5.1 Généralités sur les types mobiles

La mise en oeuvre des ressources mobiles se fait selon le protocole suivant :
1. Le type mobile 'ressource.mobile.type' est défini.
2. Une variable de type 'ressource.mobile.type' est déclarée.
3. La ressource identifiée par cette variable est crée.

Exemple :

```
1  declaration du type ressource.mobile.type
2  ressource.mobile.type ressource.id :   -- declaration de ressource.id de type ressource.mobi

3  ressource.id := MOBILE ressource.item  -- creation de la ressource d'identificateur
                                          -- ressource.id
```

Dans ce shéma 'ressource.item' est une information dont l'exploitation aboutit à la création effective de la ressource.
Si la ressource est la contrepartie mobile d'un type de données de type data.type 'ressource.item' coincide avec ce type.

Dans le cas des canaux mobiles la déclarations CHAN TYPE définit un nouveau type de ressource qui donne au compilateur les informations nécessaires pour créer le canal mobile.

Dans le cas des processus mobiles la déclaration PROC TYPE, en spécifiant la signature à laquelle doit se conformer un processus mobile, définit également un nouveau type de ressource. Mais, à l'inverse des exemples précédents, cette déclaration ne permet évidemment pas d'exhiber un processus 'ressource.item' au vu de sa seule signature. Un processus nommé qui implémente PROC TYPE sert alors à definir 'ressource.item'.
Exemple tiré des données mobiles :

```
1 MOBILE []BYTE                  --  le type MOBILE []BYTE est la contrepartie mobile
                                 --  du type []BYTE. Il ne fait pas l'objet d'une declaration

2 MOBILE []BYTE message :        -- declare message de type MOBILE []BYTE
3 message := MOBILE [10]BYTE     -- cree effectivement la ressource referencee par message
```

é

Chapitre 3

Les variables et les constantes

3.1 Les déclaration de variables

Comme il a été vu au chapitre 2 à toute ressource est associée un type.
Les variables et les constantes sont des ressources de type data.type.

Un ensemble de variables de même type se déclare en écrivant leur type commun suivi de la liste de leurs identificateurs séparés par une virgule et terminée par ' :'.
Lors de leur déclaration les variables peuvent être initialisées. Cette initialisation est annonçée par le mot clé INITIAL.
Pour nous conformer à l'usage, quand il n'y a pas d'ambiquité, nous appelerons tableau une variable de type data.table et structure une variable de type data.struct.

```
variable          => data.type {1, variable.id} :
                  => INITIAL data.type variable.id IS initial    :

data.type         => primitive.type
                  => data.table  | data.struct
variable.id       => m.identificateur
```

3.1.1 Déclaration de variables sans initialisation

```
variable          => data.type {1, variable.id} :
```

Exemples :

```
INT16 x, y, z       :  -- declare 3 variables x,y,z  de type  INT
BYTE bt             :  -- declare 1 variable bt de type   BYTE
BOOL b,c            :  -- declare 2 variables b ,c de type  BOOL

[256]BYTE buffer    :   -- une variable buffer   de type [256]BYTE
[5]INT tabx, taby   :   -- deux variables tabx, taby  de type [5]INT
[3][10] REAL32 r,rt : -- deux variables r,t de type [3][10]REAL32
```

```
--  DATA TYPE MY.STRUCT est suppose declare

MY.STRUCT str       :  -- une variable str  de type MY.STRUCT
[10]MY.STRUCT tstr  :  -- une variable tstr de type [10]MY.STRUCT
```

3.1.2 Déclaration de variables avec initialisation

```
variable      => INITIAL data.type variable.id IS initial  :

initial       =>  expression | constructor | table

constructor   => [index = base FOR compte | expression ]
table         => [{1, expression}]
```

Exemples d'initialisation de variables de type primitif.

```
INITIAL BOOL encore IS TRUE           :
INITIAL BYTE char IS #2E              :
INITIAL REAL64   perimetre IS 435.897  :
```

Un tableau s'initialise soit en listant (table) ses éléments, soit (constructeur) en exécutant une boucle qui assigne la valeur d'une expression à ses éléments .Dans ce dernier cas index est une variable de type INT . base et compte sont des expressions de type INT , le plus souvent des constantes.

Exemple :

```
INITIAL [5]INT tab2 IS [0,1,(2*3)+1,3,4]        : -- table
INITIAL [5]INT tab3 IS [i=0 FOR 5 | (5*i) + 1 ] : -- constructeur
```

Pour les tableaux de type []BYTE l'écriture d'une déclaration initialisée de type table est simplifiée si ses elements sont des caractéres ascii.Par exemple :

```
INITIAL [5]BYTE string IS ['a','b','c','d','e']  : -- s'ecrit  plus simplement
INITIAL [5]BYTE string IS "abcde"                : -- ecriture abregee
```

Une structure s'initialise par une table :

Exemple :

```
--  ch3_1.occ
--
DATA TYPE TYPEREC
  RECORD
    INT i, j    :
    BOOL  bl    :
:

PROC main( )
  INITIAL TYPEREC tpr IS [1,2,FALSE]  :
  SKIP
:
```

Un tableau dont les éléments sont de type data.struct s'initialise par une table dont les éléments sont de type data.struct qui eux même s'initialisent par une table .

Exemple :

```
-- ch3_2.occam2
--
DATA TYPE TYPEREC
  RECORD
    INT i0, i1  :
    BOOL  bl    :
:

PROC main( )
  INITIAL [3]TYPEREC recd0 IS [[0,1,TRUE] ,[2,3,FALSE] ,[4,5,TRUE]] :
  [3]TYPEREC recd1 :
  SEQ
    recd1 := recd0
:
```

3.1.3 Les éléments de tableau

Si t est un tableau de type data.type alors $t[i]$ est une variable de type data.type pour tout i tel que $0 <= i < dim$ où dim est la dimension du tableau. L'exemple ci dessous initialise les éléments de tc1 et les affiche au fur et à mesure. Le premier valant 'A' , le second 'B' , etc ..On remarque le cast (BYTE i) pour que l'expression (voir plus loin) 'A' + (BYTE i) soit de type BYTE en accord avec le type BYTE de $tc1[i]$.Ensuite, par affectation , tc2 est initialisé puis affiché.
L'exemple ci dessous montre , en sus de l'exemple précédent une autre façon d'initialiser un tableau en initialisant tous ses éléments comme dans les langages classiques.

```
-- ch3_3.occ
--
 #USE "course.lib"

 PROC main(CHAN BYTE scr!)
  [10]BYTE tc1, tc2:
  SEQ
    SEQ i= 0 FOR 10
      SEQ
        tc1[i] := 'A' + (BYTE i)
        scr! tc1[i]
        scr! #20
    scr! '*n'
    tc2 := tc1
    out.string(tc2,0,scr)
    scr ! '*n'
:
```

Le repérage des éléments d'un tableau multi dimentionnel suit la même règle .Par exemple dans la déclaration [4][4]INT tab : de l'exemple qui suit tab[i] est lui même un tableau de type [4]INT dont l'accès au j-eme élément est tab[i][j].L'index i varie de 0 à 3 et l'index j varie de 0 à 3 .

Le programme ci dessous affiche les entiers de 0 à 15 inclus , séparés par des blancs, saute la ligne, puis rend la main.

```
-- ch3_4.occ
--
 #USE "course.lib"

PROC main(CHAN BYTE scr!)
  [4][4]INT tab:
  SEQ
    SEQ i = 0 FOR 4
      SEQ j = 0 FOR 4
        SEQ
          tab[i][j] := (4*i)+ j
          out.int(tab[i][j], 0 , scr!)
          scr! #20
    scr ! '*n'
:
```

3.1.4 Les segments de tableaux

En Occam il est possible d'étendre la notion d'accès à un élément d'un tableau à un sous ensemble d'éléments contigüs de ce dernier .Ce sous ensemble : un segment de tableau , est également un tableau dont le type coincïde avec celui du tableau initial et la dimension est celle du nombre d'éléments extraits.
Par exemple :

```
-- ch3_5.occ
--
 PROC main(CHAN BYTE scr!)
  INITIAL [10]BYTE table IS "abcde45678" :
  [5]BYTE table.seg        :
  SEQ
    table.seg := [table FROM 2 FOR 5]
    SEQ i=0 FOR 5
      SEQ
        scr! table.seg[i]
        scr! #20
    scr! '*n'
:
```

Ce programme affiche c d e 4 5 saute une ligne et rend la main .
L'écriture [table FROM 2 FOR 5] est un tableau, segment de tableau de table. Ce segment a cinq éléments (FOR 5) extraits de table à partir du 3 ème élément de table (FROM 2). En particulier [table FROM 0 FOR 10] coïncide avec table.
Le compilateur vérifie que le segment [table FROM 2 FOR 5] est un sous ensemble valide de table.
La notion de segment de tableau s'applique aux tableaux multi-dimentionnels comme dans l'exemple ci dessous.

```
-- ch3_6.occ
--
```

```
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  [10][5]BYTE tab    :
  [5][5]BYTE sous.table :
  SEQ
    --  initialise table
    SEQ i=0 FOR 10
      IF
        i < 5
          tab[i]:= "01234"
        i >= 5
          tab[i]:= "56789"
    --
    sous.table := [tab FROM 4 FOR 5]
    --
    SEQ i=0 FOR 5
      SEQ
        out.string(sous.table[i],0,scr!)
        scr! '*n'
    scr! '*n'
:
```

Ce programme affiche les 5 éléments de sous.table qui est un segment de tab.
Ces cinq éléments sont eux même des tableaux de type [5]BYTE.
L'écriture d'un segment de tableau peut être abrégée si ce segment commence soit par l'élément 0 du tableau soit s'il se termine sur le dernier élément.
Plus précisemment :
Si tab est un tableau alors : [tab FOR n] dénote le segment [tab FROM 0 FOR n].
Si tab possède m éléments [tab FROM p] dénote le segment [tab FROM p FOR (m-p)] qui est le segment commençant à l'élément p et incluant le dernier élément de tab.

Syntaxe :

```
segment           =>  [tableau.id  FROM base FOR compte ]
                  =>  [tableau.id  FROM base ]
                  =>  [tableau.id FOR compte]
tableau.id        =>  m.identificateur
```

Comme pour la syntaxe des formes répliquées base et compte doivent être de type INT.
La syntaxe , récursive, montre que l'écriture d'un segment peut être assez complexe .
Pour des raisons de lisibilité le contexte d'application de la notion de segment en dehors des exemples cités ne me semble pas raisonnable quand bien même il puisse donner des écritures licites compactes.
Soit par exemple (cité dans le manuel d'occam2.1) [[c FROM j FOR i] FOR 5].
Ceci est un exemple de segment qui sélectionne les 5 premiers éléments du tableau T qui est un segment constitué de i éléments extraits à partir de l'élément j du tableau c.
Donc, si i $>=$ 5 il s'agit en fait de [c FROM j FOR 5]!!.

3.1.5 Les éléments de structure

Si s est une structure dont la définition du type possède un champ c de type data.type alors s[c] est une variable de type data.type.
L'identificateur du champ joue, pour une structure, le même rôle que celui de l'index pour un tableau.

Exemple :

```
-- ch3_7.occ
--
 DATA TYPE DATE
  RECORD
    BYTE jour :
    BYTE mois :
    INT16 an  :
:

DATA TYPE IDENTITE
  RECORD
    [10]BYTE  nom         :
    [15]BYTE  adresse     :
    DATE      date        :
    INT64     sec.sociale :
:

PROC main()
  IDENTITE nemo  :
  [2]DATE dates  :
  SEQ
    nemo[nom] := "dupont    "
    nemo[adresse] := "10 rue azerty  "
    nemo[date][jour]:= 12
    nemo[date][mois] := 9
    nemo[date][an] := 1986
    nemo[sec.sociale] := 5642008
    --
    dates[0][jour] := 19  -- dates[0] est une variable de type DATE
    dates[0][mois] := 09
    dates[0][an]   := 1976
    --
    dates[1][jour] := 20  -- date[1] est une variable de type DATE
    dates[1][mois] := 10
    dates[1][an]   := 2009
:
```

Si , comme on l'aura remarqué , $dates[i]$ est de type DATE , l'initialisation des champs $dates[i][jour]$, $dates[i][mois]$, $dates[i][an]$ est en accord avec leur type (BYTE , BYTE , INT) .
Dans le même esprit $nemo[adresse]$ est de type $[15]BYTE$ et $nemo[adresse][i]$ est le ième BYTE de $nemo[adresse]$.

3.2 Les déclarations de constantes

3.2.1 Les littéraux

Un littéral est l'écriture d'une valeur associée à un type de données primitif. Un littéral est souvent associé à la définition d'une constante.

Exemples de littéraux :

255 fait reférence, dans le système décimal, à un entier dont le type (BYTE, INT, INT16, INT32 ,INT64) est dicté par le contexte.
#FF définit la même valeur 255 que précedemment mais en base 16. Le # caractérise les constantes hexadécimales en Occam.
'A' est l'écriture d'un BYTE et coïncide avec la valeur #41(BYTE) qui est son code ascii .
TRUE , FALSE sont les seules valeurs de type BOOL.

Quand le type lié á l'écriture d'une constante peut donner lieu à confusion , où si ce dernier doit être explicité compte tenu du contexte l'indétermination est levée par un **cast** comme il a été vu pour #41(BYTE).
Par exemple l'écriture 255(INT) nous dit qu'il ne s'agit pas d'un BYTE mais d'un INT.
De même 123.76(LONGUEUR) précise que 123.76 a pour type LONGUEUR (on suppose que LONGUEUR est définit) .
123.50 ,1.235E+2, 12350.0E-2, 0.1235E+3 sont quatre écritures différentes d'une même valeur flottante de type, par exemple, REAL32 .
Pour le type REAL32 l'exposant ne doit pas comporter plus de 2 digits et pour le type REAL64 pas plus de 3 digits.
Exemple simple portant sur l'écriture des littéraux flottants :

```
-- ch3_8.occ
--
PROC main(CHAN BYTE scr!)
  REAL32  x , y, z , t:
  SEQ
    x := 0.12350E+3
    y := 1.235E+2
    z := 12350.0E-2
    t := 123.50
    IF
      (x = y) AND (y = z)AND(z = t)
        scr! '1'
      TRUE
        scr! '0'
    scr! '*n'
:
```

Ce programme affiche 1 sur le terminal .
NOTA : En général tester si 2 variables flottantes sont égales n'a pas de sens. Ici on a le cas particulier de 4 écritures différentes donnant le même code interne.

```
litteral        => number |casted.number | booleen
```

```
casted.number   => number(type.primitif)
booleen         => TRUE | FALSE
number          => integer | byte | real
integer         => s.digit | #s.hex.digits
byte            => integer | 'caractere'
real            => digits.digits | digits.digits.E exponent
exponent        => +digit.sdigit | -digit.sdigit
s.digit         => digit s.digit | vide
s.hex.digit     => hex.digit s.hex.digit | vide
hex.digit       => digit | A | B | C | D | E | F
caractere       => ABCDEFGHIJKLMNOPQRSTUVWXYZ
                => abcdefghijklmnopqrstuvwxyz
                => digit
                => ! `` ' \& ( ) * + - / :: ; < = > ?[ ] #
                => '*n' | '*r'| '*t' |'**'
digit           => 0|1|2|3|4|5|6|7|8|9
```

Les 4 (doubles) caractères '*n','*r','*t' codent pour les caractères de controle ascii que sont respectivement (saut de ligne #0A), (retour chariot #0D), (tabulation #09).
Nota : Les caractères de contrôle peuvent aussi s'écrire '*N', '*C' , etc .
Pour faire référence au caractére '*' qui intervient dans les codes de contrôle on l'écrit '**'.

3.2.2 Les constantes

Une constante identifie une ressource à de type data.type. Sa valeur n'est accessible qu'en lecture.
Une constante se déclare comme une variable initialisée sauf que INITIAL est remplaçé par VAL.

Syntaxe :

```
constante     => VAL data.type  IS expression :
constante     => VAL [{?,dimension}] data.type IS table :
table         => [{1, expression}]
```

On notera que la dimension d'une constante.tableau est optionnelle et est determinée par le nombre d'éléments de table.
Dans une constante de type data.table tous les éléments de table sont de même type alors que dans une constante de type data.struct le type des éléments de table doivent concorder avec le type défini par les champs de la structure.
Le syntaxe des expressions est etudiée au chapitre 4.

Exemple :

```
DATA TYPE TYPEREC
  RECORD
    INT i0, i1  :
    BOOL  b1    :
:

VAL BYTE bit.sept IS (1 << 7)           :
```

```
VAL REAL64 pi IS 3.141592653        :
VAL []BYTE message IS "hello"       : -- ecriture abregee . Voir ci dessous
VAL TYPEREC rec IS [10, -3, TRUE]   :

VAL []INT tab1 IS [0, 1, 2 , 3 , 4 ]:
VAL []TYPEREC recd IS [[0,(3*4)+1,TRUE] ,[2,3,FALSE] ,[4,5,TRUE]] :
```

Lorsque la dimension d'une constante tableau est declarée elle doit être en conformité avec le nombre d'éléments constitutifs du tableau et elle est de type INT.
Pour les constantes tableau de type BYTE on a une écriture abrégée par l'emploi des guillemets.
VAL [] BYTE message IS "hello" : dénote en fait VAL []BYTE message IS ['h','e','l','l','o'] :
Si la dimension n'est pas explicitée et que pour une raison quelconque on en ai besoin l'opérateur SIZE la détermine. (voir le chapitre 4 sur les expressions pour plus de précisions.
Par exemple (SIZE message) dans 'VAL []BYTE message IS "hello :" est une expression qui vaut 5 (INT).

Chapitre 4

Les expressions

Une expression, aprés évaluation, donne naissance à une valeur de type data.type.

Une expression se construit à l'aide d'opérateurs et d'opérandes.
Le type de l'expression est le type commun à tous les opérandes qui interviennent dans son écriture. Les opérateurs doivent agir en conformité avec le type commun aux opérandes. Si des ambiguités quand au type de certains opérandes peuvent exister ces ambiguités doivent être levées par des cast. Des opérateurs de conversion de type sont également parfois nécessaires.
Les opérandes sont soit des constantes soit des variables, soit des expressions entre parenthèses soit enfin des valeurs de fonctions.
Les opérateurs sont soit monadiques (portent sur un seul opérande) soit diadiques (portent sur deux opérandes) .
Attention : Tous les opérateurs ont même priorité .
Les ambiguités qui résultent de cette absence de priorité doivent être levées par le jeu des parenthèses.

Syntaxe :

```
expression => monadic.op operand
           => operand dyadic.op operand
           => type.conversion
           => operand

operand    =>  element   | (expression) | fonc.instance
element    =>  litteral | constante.id |variable.id

monadic.op => - | MINUS | BITNOT | NOT
dyadic.op  => + | - | * | / | \  REM | PLUS | MINUS | TIMES
           => /\ BITAND| \/ BITOR | >> | << | ><
           => = | <> |  < | > | >= | <= |  AFTER | BYTESIN
```

Note :
\ et REM sont deux notations équivalentes pour le reste de la division.
~ et BITNOT sont deux notations équivalentes pour le 'bitwise not'.
/\ et BITAND sont deux notations équivalentes pour le 'bitwise and'.
\/ et BITOR sont deux notations équivalentes pour le 'bitwise or'.

Par la suite nous désignerons toujours les opérateurs bit à bit par leur identificateur.

Exemples :

```
-5(INT)              -- un litteral de type INT precise par un cast.
mon.tableau[3]
x
(REAL32 u)           -- conversion : u est un INT converti en flottant sur 32 bits
6 * entier[3]        --entier est un tableau de type []INT
NOT FALSE
x << 2
(x*y) + 3
(((a+b) * (c+d)) + (e-6)) / 5
t* incr(x)           -- incr() est une fonction qui
                     -- retourne une valeur de meme type que t
```

4.0.3 Les opérateurs arithmétiques

Ce sont les opérateurs + , - , * , / , REM .
Tous ces opérateurs sont valides pour tous les types d'entiers (de BYTE a INT64) ainsi que pour les flottants . Concernant les entiers l'opérateur / est celui de la division entiére. REM exprime le reste d'une division qu'elle soit entiére ou non.
Pour tous les entiers et les flottants on a la relation :
$\forall x \quad \forall y \quad ((x/y) * y) \ + \ (x \ REM \ y) \ = \ x$
L'opérateur - (moins) est aussi monadique et est tel que $(-x) \ + \ x \ = \ 0$
pour tout x entier ou flottant.

4.0.4 Les opérateurs sur les bits

Ce sont les opérateurs BITAND , BITOR, >< , BITNOT qui opérent bit à bit sur tous les bits d'un entier.
Soient x = (x_{n-1} , ..x_k,.. x_0) et y = (y_{n-1} , ..y_k,.. y_0) deux entiers codés sur n bits alors :
Le bit de rang k de (x BITAND y) vaut 1 ssi x_k et y_k valent 1
Le bit de rang k de (x BITOR y) vaut 1 ssi x_k vaut 1 ou si y_k vaut 1
Le bit de rang k de (x >< y) vaut 1 ssi x_k et y_k valent soit (0 et 1) soit (1 et 0)
Le bit de rang k de (BITNOT x) vaut 1 si x_k vaut 0 et vaut 0 si x_k vaut 1 .

4.0.5 Les opérateurs de décalage

Ce sont les opérateurs » et « . Ils opérent sur tous les entiers.
Si x est un entier codé sur n bits (x « k) opére k décalages logiques gauche sur x pour 0 <= k < n avec k de type INT.
Les k bits de poids fort de x sont perdus et des zéros sont poussés sur les k bits de poids faible .
(x « 0) = x et (x « n) = 0 . En particulier (1 « k) code 2^k si 0 <= k < (n-1) (1 est codé sur n bits).
Si x est codé sur n bits alors x BITAND (1 « k) vaut 2^k si le bit de rang k x_k de x vaut 1 et zéro sinon .

Si x est un entier codé sur n bits (x » k) opére k décalages logiques droits sur x pour $0 <= k < n$ avec k de type INT.
Les k bits de poids faible de x sont perdus et des zéros sont poussés sur les k bits de poids fort .

4.0.6 Les opérateurs arithmétiques modulo

Ce sont les opérateurs PLUS , MINUS , TIMES .
Ces opérateurs n'affectent que les seuls entiers et opérent modulo 2^n ou n est le nombre de bits sur lequel l'entier est codé (n = 8, 16, 32 , 64) .
x et y étant codés sur n bits on a :
x PLUS y = reste de la division entière de (x+y) par 2^n .
Sur le groupe des entiers modulo 2^n l'inverse inv(x) de x vaut ((BITNOT x) PLUS 1) et vérifie donc x PLUS inv(x) = 0 .
MINUS est défini par x MINUS y = x PLUS inv(x) .
Finalement TIMES est le produit modulo 2^n .
Sur les entiers signés les opérateurs PLUS et MINUS coincident avec + et - partout où ces opérateurs fournissent des résultats valides (ne donnant pas lieu a un dépassement de capacité). Le petit programme qui suit permet de fixer les idées en considérant les bytes comme des entiers signés sur 8 bits (analogue au type char de 'C(') .

```
-- ch4_1.occ
--
BYTE FUNCTION inverse(VAL BYTE x) IS ((BITNOT x) PLUS 1):

VAL BYTE bit7 IS (1 << 7):  -- pour tester le bit de signe

PROC main(CHAN BYTE scr!)
  BYTE b0 , b1, b3, b2  :
  BYTE m0, m1, m2       :
  BYTE t0 , t1          :
  SEQ
    b0 := 25
    b1 := inverse(b0)    -- pourrait etre affiche -25
    --b3 := b0 + b1       -- b3 = 256 > 255 =>  Kroc range error
    b3 := b0 PLUS b1
    b2 := b0 PLUS 1
    out.byte(b1,0,scr!)   -- 231 car 231 + 25 = 256 = 0
    scr! ' '
    out.byte(b3,0,scr!)   -- 0 car 231 + 25 = 0
    IF
      (b1 BITAND bit7 ) <> 0
        out.string(" *nb1 est negatif", 0, scr!)
      TRUE
        out.string(" *nb1 est positif ou nul ", 0 , scr!)
    scr! '*n'
    m0 := 5
    m1 := m0 MINUS 10         -- 251 = 5 + 246
    out.byte(m1,0,scr!)       -- 10 + 246 = 256 = 0
    scr! ' '
    m2 := m0 PLUS inverse(10) --  251
    out.byte(m2,0,scr!)
```

```
    scr! '*n'
    t0 := 100
    t1 := t0 TIMES 3
    out.byte(t1,0,scr!)        -- 44 car  300 = 256 + 44
    scr! '*n'
:
```

Affiche au terminal :

```
231 0
b1 est negatif
251 251
44
```

4.0.7 Les opérateurs booléens

Ce sont AND , OR , NOT . Ils sont définis par les relations bien connues :
Si x et y sont deux variables de type BOOL alors :
x AND y a pour valeur TRUE ssi x vaut TRUE et y vaut TRUE.
x OR y a pour valeur TRUE ssi au moins un des deux opérandes x ou y vaut TRUE .
(NOT x) est TRUE ssi x a pour valeur FALSE .
AND et OR sont des opérateurs associatifs ainsi , par exemple, l'expression :
(x AND y AND z AND t) est valide et ne nécessite pas de plus de parenthèses.

4.0.8 Les opérateurs relationnels

Ce sont = égal , <> différent , < inférieur strict , > supérieur strict , <= inférieur ou égal , >= supérieur ou égal .
Il sont définis sur des opérandes entiers ou flottants .Rappelons que sur des flottants = et <> n'ont en général pas de sens à cause du codage spécifique à ce type de données.
Ces opérateurs fournissent une valeur de type BOOL .

4.0.9 Quelques opérateurs spécifiques

MOSTNEG et MOSTPOS

```
expression => MOSTPOS data.type | MOSTNEG data.type

exemples :

MOSTPOS BYTE
MOSTPOS INT32
MOSTNEG INT16
```

Ces opérateurs sont définis sur les types entiers signés INT , INT32 , etc .. ou non (BYTE) . Les entiers étant codés sur n bits :
MOSTPOS vaut 2^7 soit 255 sur les BYTE et $2^{n-1}-1$ sur les entiers signés sur n bits .
MOSTNEG vaut 0 sur les BYTE et -2^{n-1} sur les entiers signés sur n bits .

L'opérateur SIZE

Cet opérateur est utilisé pour fournir la dimension d'un tableau et délivre une valeur de type INT.

```
expression => SIZE tableau

Soient les declararations :
[5]REAL32 tfloat :
VAL []BYTE message IS "hello world"  :

SIZE tfloat  vaut 5
SIZE message vaut 11
```

L'opérateur AFTER

AFTER opère sur les entiers signés ou non .On l'utilise souvent en conjonction avec un timer pour exprimer un délai (cf le chapitre 5.3 sur le timer).
AFTER est un opérateur modulo à valeur un booléen.
Soient a et b deux entiers codés sur n bits tels que $a \neq b$.
$[0, 2^n] = [b, b + (2^{n-1} - 1] \cup [b + 2^{n-1}, (b - 1)]$.
Si $a \in [b, b + (2^{n-1} - 1)]$ alors (a AFTER b) = TRUE .
Si $a \in [b + 2^{n-1}, (b - 1)]$ alors (a AFTER b) = FALSE.
On voit que sur les entiers signés ((a AFTER b) = TRUE) exprime que :
a $<= b + 2^{n-1} - 1$ $<=>$ a MINUS b $<=$ ($2^{n-1} - 1$) $<=>$ (a MINUS b) > 0 .
Pour les BYTE qui sont non signés les conclusions sont identiques dans le sens que le bit 7 doit être nul. (le bit 7 nul d'un byte permet d'interpréter ce dernier comme positif dans une convention signée).
Pour un byte on a donc :
(a AFTER b) = TRUE $<=>$ (a MINUS b) BITAND (1«7) = 0

4.0.10 Les opérateurs de conversion

Comme il a été dit en préambule à ce chapitre tous les opérandes d'une expression doivent être de même type .
S'il y a ambiguité sur un littéral cette dernière est levée par un cast.

Exemple :

```
#FF(INT32) -- declare que #FF est un INT32
```

Conversions entre entiers

La conversion d'un entier codé sur n bits vers un entier codé sur m bits tels que $n < m$ se fait sans problème en grandeur et en signe.
Si $n > m$ la conversion s'opère sous réserve que la grandeur à convertir puisse l'être sans créer de dépassement de capacité.

Syntaxe :

```
conversion.entier => (type.primitif  variable)

exemples :
INT 16 x , y :
INT64 u , v :

x := -1
u := (INT64 x)  -- u vaut -1
v := #FFFF -- v en tant que INT64 est  > 0
y := (INT16 v)  -- depassement de capacite
```

4.0.11 Conversion entiers , flottants

La conversion d'un entier vers un flottant ne pose pas de probléme si la valeur de l'entier peut être prise en charge par le format flottant .Les problémes simples d'overflow sont détectés à la compilation.

Syntaxe :

```
conversion entier.flottant => (type.flottant TRUNC entier)
type.flottant              => REAL32 | REAL64

exemple:

INT16 x   :
REAL32 r  :

r := (REAL32 TRUNC x)
```

La conversion d'un flottant vers un entier , toujours sous les réserve de compatibilités de codages peut être soit tronquée , soit arrondie .
La troncature (TRUNC) délivre la valeur de la partie entiére du flottant. L'arrondi (ROUND) délivre la valeur de l'entier immediatement supérieur si la partie décimale du flottant est strictement supérieure à 0.5 sinon l'arrondi équivaut à une troncature .

Syntaxe :

```
conversion flottant.entier => (type.entier arrondi flottant)
type.entier                => BYTE | INT16 | INT | INT32 | INT64
arrondi                    => ROUND | TRUNC

INT16 x  :
REAL32 r :

r := 34.51
x := (INT16 ROUND r)  -- x vaut 35
x := (INT16 TRUNC r)  -- x vaut 34
--
r := 34.50
x := (INT16 ROUND r)  -- x vaut 34
```

4.0.12 Conversion flottant , flottant

Dans le sens REAL32 vers REAL64 la conversion est automatique (moyennant l'indication de format REAL64).
Dans le sens REAL64 vers REAL32 on doit impérativement adjoindre TRUNC ou ROUND . Si les plages de codages sont compatibles entre les formats 32 bits et 64 bits TRUNC et ROUND sont équivalents. Pour plus de details sur TRUNC et ROUND nous renvoyons le lecteur au manuel de reférence d'Occam 2.1 .

Syntaxe :

```
conversion flottant.flottant => flottant32.flottant64 | flottant64.flottant32
flottant32.flottant64        => (REAL64 flottant32)
flottant64.flottant32        => (REAL32 arrondi flottant64)
arrondi                      => ROUND | TRUNC
```

Exemple :

```
REAL32 r , s :
REAL64 w     :

r := 34.51
w := (REAL64 r)
s := (REAL32 ROUND w)  -- s vaut 34.51
```

Le programme suivant reprend les exemples donnés ci dessus .
L'impression des flottants 32 bits utilise le processus course.REAL32TOSTRING() de la librairie course.
Les paramètres fournis à ce processus sont la valeur du flottant , le nombre de chiffres avant et aprés le point décimal et le canal de sortie (la console) .

```
--ch4_2.occ
--
--  Les INT sont codes sur 32 bits

#USE "course.lib"

PROC print.real32(VAL REAL32 real, VAL INT pre, post, CHAN BYTE screen!)
  INITIAL [10]BYTE buffer  IS "          "     :
  INT len :
  SEQ
    course.REAL32TOSTRING(len,buffer,real,pre,post)
    out.string(buffer,0, screen!)
:

PROC main(CHAN BYTE scr!)
  INT16 x,y          :
  INT u , v           :
  REAL32 r1, r2       :
  REAL64 w1           :
  SEQ
    --  ==============================================
    --  conversions entiers <----> entiers
```

```
    --
    out.string("*nconversion entiers entiers",0, scr!)
    x := -1
    u := (INT x)        -- u vaut -1
    out.string("*nu : ", 0, scr!)
    out.int(u,0, scr!)
    v := #FFFF         -- v est considere   > 0 comme INT
    -- y := (INT16  v)  -- depassement de capacite
    --
    --  ================================================
    --  conversions entiers <----> flottants
    --
    out.string("*nconversion entiers flottants",0, scr!)
    x := 25690
    r1 := (REAL32 TRUNC x)
    out.string("*nr1 : ",0,scr!)
    print.real32(r1,4,2, scr!)
    --
    r1 := 34.51
    x := (INT16 TRUNC r1)
    out.string("*nTRUNC 34.51 : ", 0, scr!)
    out.int((INT x),0, scr!)
    x := (INT16 ROUND r1)
    out.string("*nROUND 34.51 : ", 0, scr!)
    out.int((INT x),0, scr!)
    --
    r1 := 34.50
    x := (INT16 ROUND r1)
    out.string("*nROUND 34.50 : ", 0, scr!)
    out.int((INT x),0, scr!)
    --
    r1 := (REAL32 TRUNC x)
    out.string("*nr1 : ",0,scr!)
    print.real32(r1,3,2, scr!)
    --
    --  =================================================
    --  conversions flottants <----> flottants
    out.string("*nconversion flottants flottants",0, scr!)
    r1 := 34.51
    out.string("*nr1 : ",0,scr!)
    print.real32(r1,3,2, scr!)
    w1 := (REAL64 r1)
    r2 := (REAL32 ROUND w1)
    r2 := (REAL32 TRUNC w1)  -- meme resultat qu ci dessus
    out.string("*nr2 : ",0,scr!)
    print.real32(r2,3,2, scr!)
    scr! '*n'
:
```

4.0.13 Exercice de récapitulation

Dans le programme qui suit on considére les INT16 comme non signés et on leur affecte le type U16 . Les processus read.U16() et print.U16() permettent de lire (resp écrire)

un entier de type U16. On y étudie la fonction greater() telle que $\forall x$ de type U16 $\forall y$ de type U16 greater(x,y) = TRUE ssi x > y .
Pour ce faire si x = (x_15 , ..x_k,.. x_0) et y = (y_15 , ..y_k,.. y_0) sont deux variables de type U16 on a x > y ssi $\exists k >= 0$ tel que les bits de rang 15 à (15-k) de x et de y sont égaux et $x_k = 1$ et $y_k = 0$.

```
-- ch4_3.occ
--
-- read.U16() permet d'entrer une valeur hexa X :  #0000 <= X <= #FFFF
-- print.U16() imprime un U16

#USE "course.lib"

--  a AFTER b = TRUE <=> (a MINUS b) < (MAX MINUS a) PLUS b
--  MAX = 2^16 -1
--  u < v <=>  v succede a u dans [0 , MAX] <=> greater(v , u)

DATA TYPE U16 IS INT16:

VAL U16 MAXU16 IS (MOSTPOS U16) :
VAL U16 MAX IS ((1<< 16)MINUS 1):

U16 FUNCTION inv(VAL U16 x) IS ((BITNOT x) PLUS 1):


BOOL FUNCTION greater(VAL U16 x, y)
  --
  -- retourne TRUE ssi x > y
  --
  BOOL result, encore     :
  INT i                   :
  U16 sx , sy             :
  VALOF
    SEQ
      i := 15
      encore := TRUE
      result := FALSE
      WHILE encore
        SEQ
          sx := x BITAND (1<< i)
          sy := y BITAND (1<< i)
          IF
            sx = sy
              IF
                i > 0
                  i := i-1
                TRUE
                  encore := FALSE
            sx <> sy
              SEQ
                IF
                  sx <> 0
                    result := TRUE
                  sx = 0
                    result := FALSE
```

```
            encore := FALSE
    RESULT result
:

PROC print.bool(VAL BOOL booleen, CHAN BYTE screen!)
  IF
    booleen
      out.string("TRUE", 0, screen!)
    NOT booleen
      out.string("FALSE",0, screen!)
:

PROC print.U16(VAL U16 x, CHAN BYTE screen!)
  [4]BYTE  result   :
  U16 divd , reste  :
  SEQ
    SEQ i=0 FOR 4
      result[i] := 0
    divd := x
    SEQ i=0 FOR 4
      SEQ
        reste := divd BITAND #F
        IF
          reste < 10
            result[3-i] := '0' + (BYTE reste)
          reste >= 10
            SEQ
              reste := reste - 10
              result[3-i] := 'A' + (BYTE reste)
        divd := (divd >> 4)
    --
    screen! '#'
    SEQ i=0 FOR 4
      screen! result[i]
:


U16 FUNCTION read.U16(VAL []BYTE str)
  U16 result   :
  VALOF
    SEQ
      result := 0
      SEQ i=0 FOR 4
        SEQ
          IF
            str[i] < 'A'
              result := result PLUS ((U16 (str[i] - '0')) << (4*i))
            TRUE
              result := result PLUS ((U16 ((str[i] - 'A')+ 10)) << (4*i))
    RESULT result
:

PROC main( CHAN BYTE scr!)
  U16 x , y , z , t         :
```

```
  U16 min, max , diff       :
  BOOL bool0 ,gt            :
  SEQ
    print.U16(MAX, scr!)
    scr! '*n'
    x := read.U16("FFFF")      --   #FFFF
    y := MAXU16                --   2^(15)-1 = #7FFF
    t := #7FFF(U16)            --   t = y
    print.U16(x,scr!)
    scr! '*n'
    out.int((INT x),0, scr!)  -- affiche -1
    scr! ' '
    IF
      y = t
        print.U16(y,scr!)
      TRUE
        print.U16(t,scr!)
    scr! '*n'
    -- z := y + 4              -- kroc application error , stopped
    z := y PLUS 4
    print.U16(z,scr!)          -- #8003
    scr! ' '
    t := z PLUS inv(z)
    print.U16(t,scr!)
    scr! ' '
    t := inv(z)                -- #7FFD
    print.U16(t,scr!)
    scr! '*n'
    --
    out.string("*netude de after*n",0, scr!)
    t := #200           --  arbitraire
    z := t PLUS MAXU16  -- la plus grande valeur possible pour z
    bool0 := z AFTER t
    out.string("*nz AFTER t : ",0, scr!)
    print.bool(bool0, scr!)

    min := z MINUS t
    max := (MAX MINUS z) PLUS t
    diff := max MINUS min
    out.string("*nmin := z MINUS t : ",0, scr!)
    print.U16(min, scr!)
    out.string("*nmax := (MAX MINUS z) PLUS t : ",0, scr!)
    print.U16(max, scr!)
    gt := greater(max, min)
    out.string("*ngreater(max,min) : ", 0, scr!)
    print.bool(gt, scr!)
    scr! '*n'
    x := min BITAND (1<< 15)
    print.U16(x, scr!)
    scr! '*n'
:
```

Chapitre 5

Les processus primitifs

Introduction

Un processus exécute des actions conformément à un algorithme et possède ses propres ressources. Les processus interagissent et se synchronisent en lisant où en écrivant dans des canaux.

En occam-π les barrières sont des moyens supplémentaires de synchronisation. Tous les processus occam sont construits à partir d'un nombre réduit de processus qualifiés de primitifs. Ces constructions donnent naissance aux processus composés répartis en trois catégories de processus :
Les processus séquentiels (SEQ , WHILE , IF, CASE) avec ou sans rupture de séquence.
Les processus parallèles (PAR) associés à la mise en oeuvre de la concurrence.
Les processus alternatifs (ALT) associés à la gestion du non déterminisme.

Des textes de processus peuvent être nommés. On parle alors de processus nommés et de fonctions. Les instances de processus nommés ou de fonctions sont des processus. Le code d'un processus nommé déclaré MOBILE peut être transféré dans un canal puis être executé à sa réception.
Les processus cif 'proc.cif' sont en relation avec l'interfacage d'occam-π et le langage C.

```
processus           => proc.primitif | proc.standart
                    => proc.instance | fonc.instance
                    => proc.mobile | proc.cif


proc.primitif       => action | SKIP | STOP
proc.standart       => proc.sequentiel | proc.parallele | proc.alternatif
```

Les processus primitifs sont respectivement l'affectation, les communications à travers des canaux, la lecture d'un timer, la synchronisation sur une barrière et finalement SKIP et STOP.

```
proc.primitif      => action | SKIP | STOP
action             => affectation | communication
                   => synchronisation |  lecture.timer
```

5.1 Affectation

Une affectation met en jeu une expression dont la valeur, après évaluation, est transférée à une variable.
Syntaxe :

```
affectation     => {1, variable.id} := {1, expression}
```

Important : La variable et l'expression doivent être du même type (data.type).
Une affectation multiple assigne plusieurs valeurs à plusieurs variables.
Dans ce dernier cas les affectations se font dans l'ordre des écritures.
Exemples :

```
VAL [ ]BYTE message IS "0123456789":
[10]BYTE s  :
INT x , y   :

s   := message

x,y := 10 , 20
x   := 2*(x+y)
x,y := y,x -- permutte les valeurs de x et de y
```

Tous les types 'data.type' sont concernés par l'affectation. L'exemple suivant montre une affectation sur des types data.struct.

```
-- ch5_1.occ
--
#USE "course.lib"

DATA TYPE MY.STRUCT
  RECORD
    BYTE b0 , b1     :
    [10]BYTE  str    :
:

PROC main(CHAN BYTE scr!)
  VAL [ ]BYTE message IS "0123456789"       :
  VAL MY.STRUCT mstr2 IS ['A', 'B', message]:
  MY.STRUCT mstr1 :
  SEQ
    mstr1 := mstr2
    out.string(mstr1[str], 0, scr!)
    scr! '*n'
:
```

A l'écran s'affiche 0123456789 suivi d'un saut de ligne.

5.2 Communication

Les canaux sont le moyen privilégié par lequel des processus concurrents échangent de l'information et, se faisant, se synchronisent. L'information n'est pas bufferisée et sa

structure est soumise à un protocole.
L'un des processus écrit dans un canal tandis que l'autre lit dans ce même canal. Le premier des processus prêt soit à lire soit à écrire est suspendu. Lorsque le second est lui aussi prêt alors le transfer d'information s'éffectue et les processus poursuivent leur exécution.
Ainsi la communication de deux processus à travers un canal implémante un rendez vous entre ces deux processus.

5.2.1 Déclarer un canal

Syntaxe :

```
ressource       => canal
canal           => {?, shared} channel.type {1, canal.id} :
channel.type    => CHAN protocole.id
                => [{?, expression}]CHAN protocole.id

shared          => SHARED! | SHARED? | SHARED
```

En occam-π un canal peut être partagé, en lecture, en écriture, en lecture et écriture.
L'étude des canaux partagés est faite au chapitre 11.

Exemples :

```
CHAN BYTE         b.canal          :  b.canal est un canal de protocole BYTE
CHAN MY.PRT       channel          :  channel est un canal de protocole MY.PRT
CHAN [10]INT      canal1, canal2   :  canal1 et canal2 sont deux canaux de protocole [10]INT
[10]CHAN REAL32   ctr.canal        :  ctr.canal est un tableau de 10 canaux de protocole REAL32

SHARED! CHAN INT chan.int          :  chan.int est un canal partage en ecriture de protocole INT
```

La portée de la déclaration d'un canal est limitée au processus parallèle PAR, situé au même niveau d'indentation, qui lui succède dans l'ordre de l'écriture.

Le programme qui suit présente une communication très simple entre deux processus concurrents A et B qui échangent un byte à travers le canal m.canal de protocole BYTE.
Le processus A écrit le caractere 'A' dans m.canal . Le processus B lit la valeur reçue dans la variable b puis l'écrit dans le canal scr (de protocole BYTE) lié à l'écran.

```
-- ch5_2.occ
--
PROC main(CHAN BYTE scr!)
  CHAN BYTE m.canal :   -- declare m.canal
  PAR
    --     processus A
    SEQ
      m.canal! 'A'      -- ecriture de 'A' dans m.canal
      SKIP

    --     processus B
```

```
    BYTE bt  :
    SEQ
      m.canal? bt        -- lecture de m.canal. La valeur lue est dans bt
      scr! bt            -- ecrit la valeur lue a l'ecran
      scr!'*n'           -- ecrit un saut de ligne a l'ecran
:
```

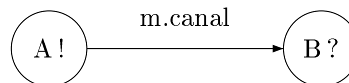


FIGURE 5.1 – A écrit dans m.canal , B lit dans m.canal

Un canal en Occam2.1 est uni-directionnel. De plus il faut deux canaux si l'on veut établir une connection bi-directionnelle entre deux processus. En occam-π un canal mobile qui regroupe en général un ensemble de canaux traditionnels permet de résoudre le problème de la liaison bi-directionnelle.

5.2.2 La communication

Une communication à travers un canal met en jeux deux processus concurrents . L'un lit le canal, l'autre écrit dans le canal.

```
commnication         => lecture.canal | ecriture.canal
```

La lecture d'un canal

Syntaxe :

```
lecture.canal        => lecture.simple
                     => lecture sequentielle
                     => lecture.variable

lecture.simple       => canal.id? input.item
input.item           => variable
                     => variable:: variable

lecture.sequentielle => canal.id? {1; input.item}

lecture.variable     => canal.id? CASE
                                  {selecteur}
selecteur            => tag
                     => tag ; {1, input.item}

canal.id             => m.identificateur
tag                  => m.identificateur
```

Si input.item est réduit à une seule variable cette dernière est de type data.type.
Si input.item est constitué de deux variables séparées par ' : :' la première variable est de type type.primitif et la seconde de type [ ]data.type.
Une lecture simple se fait sur un canal auquel est associé un protocole simple.(cf le paragraphe 2.2.1 du chapitre 2).

Exemple :

```
CHAN INT ca      :  -- le protocole de ca est INT
CHAN [10]BYTE cb :  -- le protocole de cb est [10]BYTE

exemples de lectures associees a ces protocoles

ca? va   -- va est de type INT
cb? vb   -- vb est de type [10]BYTE
--
--
PROTOCOLE PROTSIMPLE IS INT::[]BYTE :
CHAN PROTSIMPLE cc :

exemple de lectures associees a ce protocole

cc? size:: buffer      -- size est un INT buffer est un []BYTE
--
--
PROTOCOL PROTSEQ IS REAL32 ; INT ; INT::[]BYTE :
CHAN PROTSEQ cd  :     -- le protocole de cd est PROTSEQ

exemple de lectures associees a ce protocole

cd? r; i ; len::buffer -- r est un REAL32, i un INT , len un INT et buffer un []BYTE
```

Un exemple complet traitant les lectures associées aux protocoles variables sera donné au paragraphe 2.2.3

L'écriture d'un canal

A chaque mode de lecture canal : simple , séquentielle ou variable, correspond un mode d'écriture canal.

Syntaxe :

```
ecriture.canal        => ecriture.simple
                      => ecriture sequentielle
                      => ecriture.variable

ecriture.simple       => canal.id! output.item
output.item           => expression | expression :: variable

ecriture.sequentielle => canal.id! {1; output.item}

ecriture.variable     => canal.id! tag ; {1; output.item}
```

```
canal.id             => m.identificateur
tag                  => m.identificateur
```

Si output.item est de la forme 'expression : : variable' alors variable doit être de type [ ]data.type.
Les exemples qui suivent sont le pendant des exemples présentés en lecture canal.

```
exemples d'ecritures  associees a un protocole simple

CHAN INT ca      :     -- le protocole de ca est INT
CHAN [10]BYTE cb :     -- le protocole de cb est [10]BYTE

ca! 4
cb! vb   -- vb est de type [10]BYTE
--
--
PROTOCOL PROTSIMPLE IS INT::[]BYTE :
CHAN PROTSIMPLE cc                 :   -- declare cc de protocole PROTSIMPLE
VAL []BYTE buffer IS "hello"       :

cc!  (SIZE buffer):: buffer      --  (SIZE buffer) est un INT
--
-- exemple d'ecriture associee a un protocole sequentiel
--
PROTOCOL PROTSEQ IS REAL32 ; INT ; INT::[]BYTE :
CHAN PROTSEQ cd  :               -- declare cd de protocole PROTSEQ

cd!  3.14 ; i ; (SIZE buffer)::buffer   -- i est un INT de meme que (SIZE buffer)
                                        -- buffer est est un []BYTE
```

5.2.3 Exemples complets de lectures et d'écritures

On peut sauter ces exemples en première lecture et y revenir après avoir étudié les processus séquentiels SEQ (paragraphe 6.1 du chapitre ??) et parallèles PAR (au chapitre 7).

protocoles simples

Premier exemple :
Le processus A écrit la chaine "0123456789" dans le canal cpstr de protocole [10]BYTE renommé, par déclaration de cprotocole en PSTR. Cette chaine est lue et affichée à l'écran par le processus B.

```
-- ch5_3.occ
--
#USE "course.lib"

PROTOCOL PSTR IS [10]BYTE:

PROC main(CHAN BYTE scr!)
```

```
  CHAN PSTR cpstr :
  PAR
    -- processus A
    VAL []BYTE str IS "0123456789"  :
    SEQ
      cpstr! str
      SKIP
    -- processus B
    [10]BYTE s  :
    SEQ
      cpstr? s
      out.string(s,0, scr!)
      scr! '*n'
:
```

Deuxieme exemple :
Dans cet exemple de protocole simple un enregistrement a.date de type DATE est transféré dans le canal canal. Le processus A écrit a.date et B lit sa valeur dans b.date et l'affiche à l'écran.

```
-- ch5_4.occ
--
#USE "course.lib"

DATA TYPE DATE
  RECORD
    BYTE jour, mois :
    INT16 an        :
:


PROC main(CHAN BYTE scr!)
  CHAN DATE canal  :
  PAR
    --  processus A
    VAL DATE  a.date IS [19, 09, 1985] :
    SEQ
       canal! a.date
       SKIP

    --  processus B
    DATE b.date   :
    SEQ
      canal? b.date
      out.byte(b.date[jour],0, scr!)
      scr! ' '
      out.byte(b.date[mois],0, scr!)
      scr! ' '
      out.int( (INT b.date[an]), 0 , scr!)
      scr! '*n'
:
```

Parmis les échanges que les processus concurrents pratiquent les chaines de caractères revêtent une grande importance.

Dans cet exemple le protocole STR est INT : :[]BYTE qui traduit qu'un transfert sur m.canal consiste en un entier suivi d'une chaine de caractères . L'entier défini par (SIZE message) donne la longueur de la chaine.

```
-- ch5_5.occ
--
PROTOCOL STR IS INT::[]BYTE :

PROC main(CHAN BYTE scr!)
  CHAN STR   m.canal :
  PAR
    --  processus A
    VAL []BYTE message IS "Hello World !! *n":
    SEQ
      m.canal ! (SIZE message)::message
      SKIP

    --  processus B
    [256]BYTE buffer  :
    INT length        :
    SEQ
      m.canal ? length:: buffer
      SEQ i=0 FOR length
        scr ! buffer[i]
:
```

Ce dernier exemple de protocole simple de la forme 'type.primitif : :[]data.type' prend pour data.type le type renommé STRING. A l'ecran s'affiche “hello” 4 fois

```
-- ch5_6.occ
--
DATA TYPE STRING IS [5]BYTE     :
PROTOCOL PRT IS INT::[]STRING   :

PROC main(CHAN BYTE scr!)
  CHAN PRT m.canal                :
  PAR
    -- processus A
    VAL STRING  message IS "hello"  :
    INITIAL [4]STRING  str IS [i=0 FOR 4 | message ] :
    SEQ
      m.canal! (SIZE str)::str
      SKIP

    -- processus B
    INT len       :
    [4]STRING s   :
    SEQ -- processus B
      m.canal? len::s
      -- affichage de s
      SEQ i=0 FOR 4
        SEQ
          SEQ j=0 FOR 5
            scr! s[i][j]
```

```
          scr! '*n'
:
```

protocoles sequentiels

Rapelons qu'un protocole séquentiel est défini comme une liste non vide dont les éléments sont des protocoles simples séparés par un point virgule.
Dans l'exemple qui suit le protocole STRC consiste en un INT suivi d'un INT : :[]BYTE.
Exemple :

```
-- ch5_7.occ
--
PROTOCOL STRC IS INT;INT::[]BYTE :
--
PROC main(CHAN BYTE  scr!)

  CHAN STRC canal :
  PAR
    -- processus A
    VAL []BYTE message IS "Hello World !! *n":
    VAL INT type  IS 1  :
    SEQ
      canal ! type;(SIZE message)::message
      SKIP
    --
    -- processus B
    [256]BYTE buffer   :
    INT length , pcode :
    SEQ
      canal ? pcode ;length:: buffer
      SEQ i=0 FOR length
        scr ! buffer[i]
      scr ! #30 + (BYTE pcode)
      scr ! '*n'
:
```

On peut voire l'entier pcode qui est lu en premier comme un code caractérisant le format de données en transit. Nous verrons que dans les applications il est souvent nécessaire d'indiquer le type de la trame qui transite dans un réseau .

protocoles variables

Comme on le sait un protocole variable permet de faire transiter dans un même canal des trames de formats différents. La seule contrainte est qu'un trame soit associée à un prorocole séquentiel. Les trames sont précédées d'un tag qui permet de les discriminer à la lecture. On remarque qu'un tag peut être isolé. Ce fait peut être utilisé pour marquer la fin d'une suite de transferts.
Dans le canal 'canal' de protocole PRTV sont écrits successivements : un BYTE, un couple INT ; INT , un INT : :[]BYTE et ... rien.
Ces envois sont discriminés à la réception par les tag : char, complex, string et halt.

```
ch5_8.occ
--
PROTOCOL PRTV
  CASE
    char    ; BYTE
    complex ; INT; INT
    string  ; INT::[]BYTE
    halt
:


PROC main(CHAN BYTE scr!)
  CHAN PRTV canal  :
  PAR
     --  processus A
    VAL []BYTE message IS "Hello World . . .*n":
    SEQ
      canal ! char    ;'A'
      canal ! string  ;(SIZE message):: message
      canal ! complex ; 3; 4
      canal ! halt

    --  processus B
    [256]BYTE buffer   :
    INT len , c1, c2   :
    BYTE x             :
    SEQ
      SEQ i=0 FOR 4
        canal ? CASE
          char ; x
            SEQ
              scr ! x
              scr ! '*n'
          string ; len::buffer
            SEQ i=0 FOR len
              scr ! buffer[i]
          complex ; c1;c2
            SEQ
              scr ! #30 + (BYTE c1)
              scr ! ','
              scr ! #30 + (BYTE c2)
              scr ! '*n'
          halt
            SEQ
              scr ! '**'
              scr ! '*n'
:   -- main()
```

5.3 Le timer

Les valeurs lues sur un timer sont de type **INT** et sont associées au timer du système d'exploitation de la machine.

Ces valeurs sont incrémentées à des intervalles de temps régulier et repassent cycliquement de la valeur du plus grand entier positif au plus grand entier négatif.

5.3.1 Déclarer un timer

La déclaration d'un timer est similaire à celle d'une variable.
Syntaxe !

```
ressource    => timer
timer        =>  TIMER {1, timer.id} :
```

Exemples :

```
TIMER clock          :   -- clock est un identificateur de timer
TIMER clock1, clock2 :   -- idem pour clock1 clock2
```

5.3.2 Lecture d'un timer

Syntaxe :

```
proc.primitif   => lecture.timer
lecture.timer   => timer.id? variable.id
                => timer.id? AFTER variable.id PLUS expression
```

Exemples :

```
clock? time
clock? AFTER time PLUS 100000
```

Provoque la lecture de la valeur actuelle du timer clock dans time.
La lecture du timer peut être différée par un délais en utilisant AFTER.

```
INT now :
VAL INT delais IS 1000 :
SEQ
  processus1
  clock ? now
  clock ? AFTER now PLUS delais
  processus2
```

Le processus "processus2" ne s'exécutera que passé le délais de 1000 ticks après la terminaison de "processus1" repérée par "clock ? now".
Considérons le petit programme suivant :

```
-- ch5_9.occ
--
 #USE "course.lib"

PROC main(CHAN BYTE kbd, scr, err)
  VAL INT sec IS 999900     :
```

```
  TIMER clock               :
  INT i, time1              :
  SEQ
    i := 0
    WHILE i < 10
      SEQ
        clock ? time1
        clock ? AFTER time1 PLUS sec
        scr! '.'
        scr! ' '
        flush(scr)
        i := i+1
    scr ! 'X'
    scr ! '*n'
:
```

Ce programme affiche un point toutes les secondes et se termine au bout de 10 secondes . La valeur sec = 999900 est le nombre de ticks du timer correspondant à une seconde sur ma machine.(Il est nécessaire de flusher la sortie écran à chaque affichage sinon le tampon de sortie associé à l'écran serait vidé en totalité à la fin du programme). Lors de l'étude des processus alternatifs au Chapitre 8 nous verrons une application très importante du timer dans l'implémentation d'un 'chien de garde'.

5.4 La synchronisation sur une barrière

Les barrières (de synchronisation) permettent à plusieurs processus de mettre en oeuvre un rendez vous. Le processus qui atteint le point de rendez vous en exécutant SYNC sur une barrière est suspendu tant que l'ensemble des processus enrolés sur cette barrière n'ont pas atteints leur point de rendez vous.
RESIGN permet à un processus enrolé sur une barrière de se désenroler.

5.4.1 Déclarer une barriere

Une barriere se declare comme une variable ordinaire.

Syntaxe :

```
ressource          => barriere
barriere           => BARRIER {1, barriere.id} :
```

Exemple :

```
BARRIER bar :
```

5.4.2 Synchronisation

Syntaxe :

```
synchronisation => SYNC barriere.id | RESIGN barriere.id
barriere.id     => m.identificateur
```

L'exemple suivant est caractéristique de l'utilisation d'une barrière.

```
-- ch5_10.occ
--
#USE "course.lib"


PROC affiche(CHAN BYTE ecran ,[]CHAN BYTE canal?)
  BYTE c      :
  SEQ i=0 FOR 50
    ALT i= 0 FOR 5
      canal[i] ? c
        ecran ! c

:


PROC process(VAL INT id, BARRIER ba,CHAN BYTE canal!)

  SEQ
    -- premiere partie
    SEQ i=0 FOR 5
      canal ! '0' + (BYTE id)
    SYNC ba
    -- deuxieme partie
    SEQ i=0 FOR 5
      canal ! 'A' + (BYTE id)

:


PROC main(CHAN BYTE scr!)
  [5]CHAN BYTE canal :
  BARRIER bar:
  SEQ
    PAR
      PAR i=0 FOR 5 BARRIER bar
        process(i, bar,canal[i]!)
        --
      affiche(scr, canal?)
    out.string("*nBY ...*n",0,scr!)

:
```

Dans le processus main() la barrière bar est déclarée comme une variable ordinaire .
Le processus répliqué PAR i=0 FOR 5 est suivi de BARRIER bar qui indique que les cinq processus sont enrolés sur la barrière bar qui par ailleurs est passée en paramètre à chacun d'entre eux.
Le texte de chaque processus process est composé de deux parties séparées par SYNC bar.

A l'exécution , un processus qui exécute SYNC bar est mis en attente si les autres processus n'ont pas exécutés SYNC bar de leur côté.

A l'exécution on peut voir sur le terminal :

```
40000011111222223333344444EAAAAABBBBBCCCCCDDDDDEEEE
BY ...
```

A l'entrelaçage propre à la gestion du parallélisme on voit que les cinq processus exécutent tous du code lié à la première partie avant d'exécuter du code lié à la deuxième partie.

5.5 SKIP et STOP

Les lecteurs non familiés de CSP seront intrigués par SKIP et STOP.
SKIP est un processus qui , une fois lancé, se termine sans avoir fait la moindre action.
STOP est un processus qui, une fois lancé , n'exécute aucune action et ne se termine jamais.
Si on assimile une automobile à un processus dont les actions consistent à rouler alors une voiture avec une bonne batterie mais avec la boite de vitesse cassée démarre mais ne peut pas rouler. Le comportement de cette voiture peut être assimilé à STOP.
SKIP joue un rôle important dans des processus comme IF où ALT où, dans un certain contexte, il ne faut rien faire.
Considérons les deux programmes suivants :

```
PROC main(CHAN BYTE kbd?, scr!, err!)
  SEQ
    scr ! 'A'
    scr ! '*n'
    SKIP
    scr ! 'B'
    scr ! '*n'
:
```

```
 PROC main(CHAN BYTE kbd?, scr!, err!)
  SEQ
    scr ! 'A'
    scr ! '*n'
    STOP
    scr ! 'B'
    scr ! '*n'
:
```

Le premier affiche A au terminal, saute la ligne, affiche B, saute la ligne , rend la main.
Le second affiche A au terminal, saute la ligne et rend la main en affichant le message :
KRoC : Application level error, Wptr = 0x900ad78.
Note : S'il n'y avait pas eu d'affichage de saut de ligne après affichage du 'A',alors dans le deuxième programme, le 'K' de KRoC affiché par le système l'aurait masqué.

Chapitre 6

Les processus séquentiels

Les processus séquentiels regroupent les processus SEQ, WHILE, IF , CASE .
SEQ implémante la séquence pure tandis que WHILE,IF et CASE sont associés aux ruptures de séquences et aux boucles. Le processus SEQ repliqué implémante la boucle for.

```
processus           => proc.primitif | proc.standart
                    => proc.instance | fonc.instance
                    => proc.mobile | proc.cif


proc.primitif       => action | SKIP | STOP
proc.standart       => proc.sequentiel | proc.parallele | proc.alternatif
proc.sequentiel     => proc.seq | proc.if | proc.cas | proc.case
```

6.1 Les processus SEQ

Un processus SEQ combine une liste de processus de façon telle que la terminaison de l'un d'entre eux entraine le lancement de celui qui lui succède dans la liste (si toutefois il se termine!). La terminaison du dernier processus de la liste définit la terminaison du processus séquentiel.

Exemple :

```
-- ch6_1.occ
--
PROC main(CHAN BYTE scr!)
  SEQ
    scr ! 'O'
    scr ! 'c'
    scr ! 'c'
    scr ! 'a'
    scr ! 'm'
    scr ! '*n'
:
```

Affiche Occam sur le terminal, saute une ligne, et se termine.
Syntaxe :

```
proc.seq     => SEQ {?,replicateur}
                  processus

replicateur => identificateur = base FOR compte {?,STEP pas}
base        => expression
compte      => expression
pas         => expression
```

Tous les processus constitutifs de la séquence sont alignés à deux caractères blancs au dela du S de SEQ.
Toutes les expressions qui interviennent dans la syntaxe sont de type INT.

Si la liste des processus qui constitue le processus séquentiel est vide comme l'autorise la syntaxe alors SEQ se comporte comme SKIP. Ce fait est signalé par Kroc à la compilation.

6.1.1 Les processus SEQ répliqués

Reprenons l'exemple ci dessus et supposons que les caractères à afficher soient dans un tableau .Si le nombre d'éléments de ce tableau dépasse une taille jugée importante l'écriture ci dessus devient vite impraticable .
La réplication , analogue à la boucle **for** des langages usuels, permet de remédier à cette situation.
Considérons l'exemple suivant qui constitue, sous forme répliquée, la contrepartie de l'exemple précédent :

```
-- ch6_2.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL [6]BYTE message IS ['O','c','c','a','m','*n']:
  INT n:
  SEQ
    n := 6
    SEQ i= 0 FOR n
      scr ! message[i]
:
```

Syntaxe :

```
proc.seq     => SEQ replicateur
                  processus

replicateur => identificateur = base FOR compte {?,STEP pas}
```

La base fournit la valeur initiale de la variable et compte fournit le nombre d'itérations à effectuer. L'option STEP définit la manière dont base est incrementé(par défaut base est incrementé de 1). Dans l'exemple précédent si on écrit :

```
-- ch6_3.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL [6]BYTE message IS ['O','c','c','a','m','*n']:
  SEQ i= 0 FOR 6 STEP 2
    scr ! message[i]
:
```

Alors seront affichés les caractères O , c et m et il n'y aura pas de saut de ligne.

ATTENTION : Dans occam-π 'compte' n'est plus astreint à être une constante.
Dans l'exemple cité la base vaut 0 et compte vaut 6.
Important : 'base', 'compte' et 'pas' doivent être de type INT.
Notons également que si compte vaut 0 alors le processus répliqué se comporte comme SKIP.

6.2 Les processus IF

Dans un processus conditionnel une liste de choix est présentée. Un choix est constitué d'une expression booléenne à laquelle est associée un processus.
Les expressions sont évaluées en séquence. La première qui s'évalue à TRUE conditionne l'exécution du processus qui lui est associé.
Attention : Si aucune des expressions ne s'évalue à TRUE le processus conditionnel se comporte comme STOP.
Une façon élégante de ne pas être bloqué dans un processus conditionnel est d'ajouter une condition TRUE associée au processus SKIP.
Syntaxe :

```
proc.if          => IF {?,replicateur}
                         { choix }
choix       => expression
                         processus
```

Dans 'choix' l'expression est booléenne.

Considérons l'exemple suivant :

```
-- ch6_4.occ
--
PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL INT x IS 10 :
  VAL INT y IS 5  :
  IF
    x > y
      SEQ
        scr ! '1'
        scr ! '*n'
    x < y
      SEQ
        scr ! '0'
        scr ! '*n'
:
```

La valeur initialde de x est 10, celle de y est 5 . La premiére condition étant satisfaite le premier processus séquentiel est lancé. Au terminal 1 s'affiche, il y a saut de ligne,et le système rend la main .
Si l'on change la première condition en 'x = y' aucune des deux conditions n'est satisfaite et le processus conditionnel se comporte comme STOP.

Dans l'exemple ci dessous l'exécution du processus conditionnel a pour effet de rendre immédiatement la main.

```
-- ch6_5.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL INT x IS 5   :
  VAL INT y IS 5   :
  IF
    x > y
      SEQ
        scr ! '1'
        scr ! '*n'
    x < y
      SEQ
        scr ! '0'
        scr ! '*n'
    TRUE
      SKIP
:
```

On aura noté que dans la syntaxe du choix la clause conditionnel permet d'imbriquer plusieurs processus conditionnels l'un de ces processus pouvant être un processus conditionnel répliqué.
Dans la clause 'choix.garde' le processus est indenté de deux blancs relativement au premier caractère de l'expression qui lui est associée.

6.2.1 Les processus IF répliqués

Comme pour les processus SEQ une forme répliquée est définie pour traiter un nombre fini mais important de conditions indexées . Par exemple :

```
-- ch6_6.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL [6]BYTE message IS ['O','c','c','a','m','*n']:
  IF
    IF i=0 FOR 6
      message[i]= 'X'
        SEQ
          scr ! '1'
          scr ! '*n'
    TRUE
      SEQ
        scr ! '0'
        scr ! '*n'
:
```

Dans cet exemple on voit qu'un processus conditionnel répliqué intervient en place d'un processus conditionnel comme la syntaxe le prévoit .Dans l'exemple cité le processus affiche '0' au terminal.

6.3 Les processus CASE

La sélection présente une suite d'options dont l'une est prise lorsque la valeur d'un sélecteur coïncide avec l'une d'entre elles.

Syntaxe :

```
proc.case   => CASE selecteur
                 {option}
option      => option1 | option2
option1     => {1,case.option}
                 processus
option2     => ELSE
                 processus
selecteur   => expression
case.option => expression
```

Exemple :
Les options sont 2 et 4 . Le sélecteur est x .Aucune des options n'étant satisfaites pour x qui vaut 8 l'option par défaut ELSE est choisie et X s'affiche au terminal.

```
-- ch6_7.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL INT x IS 8  :
  CASE x
    2
      SEQ
        scr !'2'
        scr !'*n'
    4
```

```
      SEQ
        scr !'4'
        scr !'*n'
    ELSE
      SEQ
        scr !'X'
        scr !'*n'
:
```

Considérons maintenant ce nouvel exemple :

```
ch6_8.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL INT x IS 8  :
  CASE x
    0,2,4,6,8
      --  x est pair
      SEQ
        scr ! 'P'
        scr ! '*n'
    1,3,5,7,9
      --  x est impair
      SEQ
        scr ! 'I'
        scr ! '*n'
    ELSE
      SKIP
:
```

Là tout l'intérèt de la sélection apparait lorque l'appartenance à un ensemble donné comme dans l' exemple cité, doit être testée.

Le sélecteur et case.option doivent être du même type qui est soit un INT soit un BYTE. Par ailleurs une seule option ELSE est permise .

6.4 Les processus WHILE

Uun processus WHILE se répète un nombre fini de fois. Chaque répétition est commandée par une expression booléenne qui doit s'évaluer à TRUE .

Syntaxe :

```
proc.while          => WHILE expression
                         processus
```

Exemple :

```
ch6_9.occ
--
```

```
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL [6]BYTE message IS ['O','c','c','a','m', '!','!''*n']:
  INT i  :
  SEQ
    i := 0
    WHILE i < 6
      SEQ
        scr ! message[i]
        i := i+1
:
```

Ce programme affiche les 6 caracteres de 0 à 5 du tableau message.

La boucle est utile lorsque le nombre de répétitions n'est pas connu à l'avance et dépend de l'évaluation de l'expression qui suit le WHILE comme dans l'exemple suivant.
Ce programme détermine la première occurence , si elle existe, d'un caractère commun à deux chaines de caractères et l'affiche sinon il affiche le caractère ?.

```
ch6_10.occ
--
 PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL [9] BYTE message1 IS ['O','c','c','a','m','-','p','i','*n']      :
  VAL [10]BYTE message2 IS ['H','e','l','l','o','p','p','p','p','*n'] :
  INT i         :
  BOOL trouve   :
  SEQ
    i := 0
    trouve := FALSE
    WHILE ((message1[i] <> '*n') OR (message2[i] <> '*n')) AND (NOT trouve)
      IF
        message1[i]= message2[i]
          trouve := TRUE
        TRUE
          SKIP
    IF
      trouve
        SEQ
          scr ! message1[i]
          scr ! '*n'
      TRUE
        SEQ
          scr ! '?'
          scr ! '*n'
:
```

Chapitre 7

Les processus parallèles

7.1 Les processus PAR

Les processus parallèles PAR (on dit aussi concurents) sont au coeur de la programmation en Occam. Le parallélisme peut résulter d'actions se déroulant sur des processeurs différents où sur le même processeur . Dans ce dernier cas on sait que les actions associées aux différents processus qui composent un processus parallèle s'entrelacent dans le temps.

Syntaxe :

```
proc.parallele => PAR {?,replicateur}
                     {processus}
```

Par exemple :

```
 PAR
   processus_1
   processus_2
   ....
   processus_n
```

Un processus parallèle se termine lorsque **tous** les processus qui le composent se sont terminés. L'ordre dans lequel ces derniers se terminent n'intervient pas .

Dans cet exemple les deux processus qui,composent le processus parallèle ne communiquent pas. Comme le modèle de concurrence associe a Occam2.1 (CSP) exclue le partage de mémoire cet exemple est de peu d'interêt car les processus doivent pouvoir coopérer en communiquant.

```
-- ch7_1.occ
--
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  SEQ
    --
    --  le processus // est lance
    PAR
      --  processus 1
      INITIAL INT i IS 3  :
      WHILE i > 0
        i := i-1
      --  processus 2
      INITIAL INT i IS 10 :
      WHILE i > 0
        i := i-1
    --  le procesus // est acheve
    --
    out.string("BY ...*n",0, scr!)
:
```

Ici les processus qui s'exécutent en paralléle communiquent à travers le canal 'canal'.

in.byte() est un processus nommé (cf le chapitre 9 sur les processus nommés) dont la fonction est de scruter le clavier et de retourner un (1 seul) caractère validé par 'ENTER'. La bibliothèque 'course' de kroc n'offre hélas pas de primitive de lecture non bloquante et sans écho du clavier. in.byte() en utilisant la fonction file.in.string() de la lirairie 'course' permet la saisie bloquante d'un caractère sans écho. Ce dernier est validé par 'enter'.Il eut été plus simple d'utiliser le processus in.byte() de 'course' mais ce processus utilise l'écran qui, de ce fait, aurait été partagé entre les deux processus. Le partage des canaux est licite en occam-π et sera étudié au chapitre 11

Le processus identifié 'processus1' en commentaires fait une requête clavier. Il transmet le résultat de cette requète a 'processus2' par le biais du canal 'canal'.Si la valeur reçue après cette requète est 'q' il se termine sinon il reste dans la boucle qui consiste à scruter le clavier.

Le processus identifié 'processus2' en commentaires lit la valeur émise par 'processus1 ' dans 'canal'. Si cette valeur est 'q' il se termine sinon il affiche cette valeur, et boucle en attente de la lecture d'une nouvelle valeur.
Cet exemple très simple montre entre autres choses comment faire terminer proprement deux processus communiquants. Ici il s'agit de la reconnaissance de la valeur 'q'.

```
-- ch7_2.occ
--
#USE "course.lib"

PROC in.byte(CHAN BYTE clavier?, BYTE result)
  [10]BYTE buffer   :
  VAL INT max IS 10 :
  INT len           :
  SEQ
    --  saisie sans echo d'une chaine de longueur len
    --  max designe le maximum de caracteres autorises
    file.in.string(buffer,len,max, clavier?)
    IF
      len <> 1
        result := 'q'
      TRUE
        result := buffer[0]
:

PROC main(CHAN BYTE kbd?, scr!)
  CHAN BYTE canal  :
  PAR
    --  processus 1
    BYTE i        :
    INITIAL BOOL encore IS TRUE :
    WHILE encore
      SEQ
        in.byte(kbd? , i)
        canal! i
        IF
          i <> 'q'
            SKIP
          TRUE
            encore := FALSE

   -- processus 2
    INITIAL BOOL encore IS TRUE  :
    BYTE j      :
    WHILE encore
      SEQ
        canal? j
        IF
          j <> 'q'
            SEQ
              scr! j
              scr! ' '
              flush(scr!)
          TRUE
            SEQ
              scr! '*n'
              encore := FALSE
:
```

7.1.1 Les processus PAR répliqués

Un processus parallèle répliqué est utilisé lorsque des processus communiquants exécutent en parallèle le même code. Typiquement ils sont utilisés lorsqu'ils constituent les noeuds d'une architecture parallèle comme les grilles , les arbres , les hypercubes etc ..

7.1.2 Circulation sur un anneau

Le programme qui suit utilise un processus parallèle repliqué et présente une des plus simple des architectures parallèles : celle d'anneau . Chacun des 4 processus qui constituent cet anneau est relié à ses deux voisins par un canal. Un tableau de canaux dont les éléments ont un indice naturellement associé à celui des processus est utilisé pour implémanter ces liaisons . Le processus d'indice 0 écrit 'Z' au processus d'indice 1 à travers le canal c[1] puis il lit l'information qui a circulé dans l'anneau , l'affiche puis se termine. Un processus d'index i $>$ 0 lit l'information sur le canal c[i] ,recopie celle ci dans le canal c[((i+1) REM 4)] , et se termine .

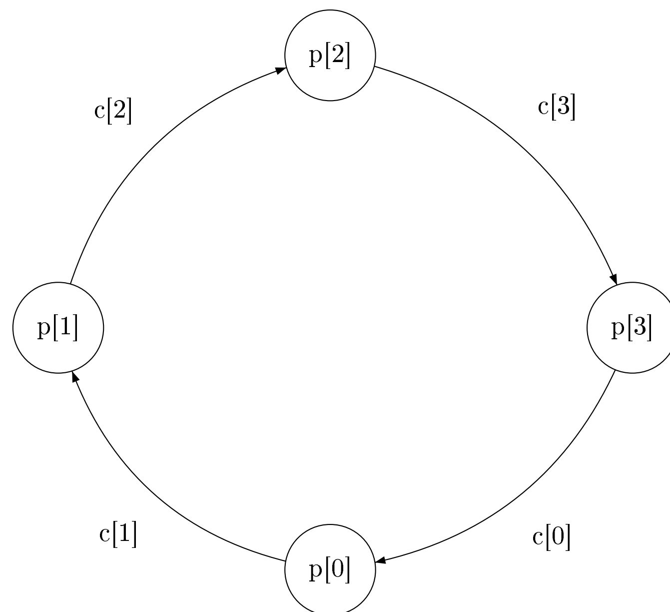


FIGURE 7.1 – Un anneau de 4 processus

```
-- ch7_3.occ
--
PROC main(CHAN BYTE scr!)
  [4]CHAN BYTE canal  :
  PAR i=0 FOR 4
    BYTE x  :
    IF
      i = 0
        SEQ
          canal[1]! 'Z'
          canal[0]? x
          scr! x
          scr! '*n'

      i <> 0
        SEQ
          canal [i]? x
          canal[(i+1) REM 4] ! x
:
```

En occam-π des priorités peuvent être affectées aux processus. Nous renvoyons à l'article 'An occam-pi Quick Reference' de P.H. Welch pour plus de précisions. Cet article se télécharge à :
`https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference`.
NB L'instruction PRI PAR issue des priorités attribuées aux processus sous occam2.1 est ignorée de Kroc.

Chapitre 8

Les processus alternatifs

8.1 Les processus ALT

En occam-π les processus alternatifs ALT sont associés à la gestion du non déterminisme.

Syntaxe :

```
proc.alternatif      => {?,PRI} ALT {?,replicateur}
                            {alternative}
alternative          => alt.guarde | alternatif
alt.guarde           => guarde
                           {processus}

guarde               => {?,boolean.expression &} lecture
                     => {?,boolean.expression &} SKIP

lecture              => lecture.canal | lecture.timer
```

Si alternative est vide ALT se comporte comme STOP.
Les gardes associées à une lecture timer (cf l'exemple du watchdog ci dessous) ou à SKIP sont prêtes dès lors que les expressions booléennes associées le sont. Un processus alternatif, lorsqu'il est activé, consulte l'ensemble de ses gardes.
Dans l'exemple qui suit les gardes sont ' b0 & canal[0] ? x' et ' b1 & canal[1] ? x'.
Une garde, ici du type lecture canal, est dite prête si les deux conditions qui suivent sont satisfaites :
1 _l'expression booléenne b0 (resp b1) qui lui est attachée s'évalue à TRUE.
2 _le canal qui lui est attaché canal[0] (resp canal[1]) est prêt en lecture.

Si aucune garde n'est prête le processus altenatif est suspendu en attente qu'au moins une d'entre elles le soit.

Si plusieurs gardes sont prêtes l'une est choisie au hasard, son canal est lu. et le processus associé est exécuté. La terminaison de ce processus définit la terminaison du processus alternatif.

Exemple :

```
-- ch8_1.occ
--
PROC main(CHAN BYTE scr!)
  [2]CHAN BYTE canal  :
  PAR
    -- processus 0
    VAL []BYTE message IS "p0p0p0p0p0+ " :
    INITIAL INT i IS 0  :
    SEQ
      WHILE message[i] <> '+'
        SEQ
          canal[0] ! message[i]
          i := i +1
      canal[0]! '+'

    -- processus  1
    VAL []BYTE message IS "p1p1p1p1p1p1+ " :
    INITIAL INT i IS 0 :
    SEQ
      WHILE message[i] <> '+'
        SEQ
          canal[1] ! message[i]
          i := i +1
      canal[1]! '+'

    -- processus recepteur
    BYTE x          :
    BOOL b0, b1     :
    SEQ
      b0, b1  := TRUE, TRUE
      WHILE (b0 OR b1)
        ALT
          b0 & canal[0]? x    -- premiere garde
            SEQ
              scr !x
              IF
                x = '+'
                  b0 := FALSE
                TRUE
                  SKIP
          b1 & canal[1] ? x    -- deuxieme garde
            SEQ
              scr ! x
              IF
                x = '+'
                  b1 := FALSE
                TRUE
                  SKIP
      scr! '*n'
:
```

8.1.1 Les processus alternatifs répliqués

Si les garde et les processus qui leur sont associés partagent le même texte et ne diffèrent que par la valeur d'un paramètre entier alors on a une version compacte du processus alternatif comme celles que l'on a vues pour les processus séquentiels où parallèles. Dans le cas de notre exemple la forme répliquée du processus alternatif associé au processus récepteur est :

```
-- ch8_2.occ
--
PROC main(CHAN BYTE scr!)
  [2]CHAN BYTE canal  :
  PAR
    -- processus 0
    VAL []BYTE message IS "popopopopo+ " :
    INITIAL INT i IS 0  :
    SEQ
      WHILE message[i] <> '+'
        SEQ
          canal[0] ! message[i]
          i := i +1
      canal[0]! '+'

    -- processus  1
    VAL []BYTE message IS "pipipipipipipipipipipip+ " :
    INITIAL INT i IS 0 :
    SEQ
      WHILE message[i] <> '+'
        SEQ
          canal[1] ! message[i]
          i := i +1
      canal[1]! '+'

    -- processus recepteur
    BYTE x          :
    INITIAL [2]BOOL b IS [TRUE, TRUE]   :
    SEQ
      WHILE (b[0] OR b[1])
        ALT   k=0 FOR 2
          b[k] & canal[k]? x
            SEQ
              scr !x
              IF
                x = '+'
                  b[k] := FALSE
                TRUE
                  SKIP

      scr! '*n'
:
```

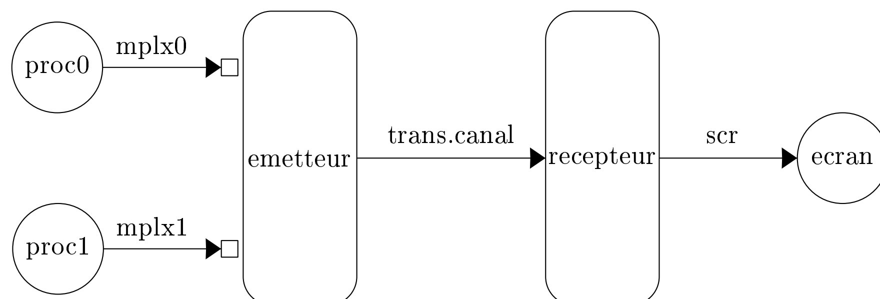


FIGURE 8.1 – multiplexage

8.1.2 Multiplexage

Multiplexer des données issues de sources différentes dans un même canal est facile à implémenter grace aux processus alternatifs. Dans l'exemple qui est présenté deux processus émettent chacun un message. Les deux messages transitent, entrelaçés, à travers le canal trans.canal. Le processus alternatif emetteur() permet cet entrelaçage en lisant les données émisent par les processus sur les deux canaux mplx[0], mplx[1] et en les écrivant sur trans.canal .
Le processus recepteur() lit les données émises par emetteur() sur trans.canal et les ventile dans deux buffers suivant que ces données sont issues soit du processus d'index 0 , soit du processus d'index 1.
Cette identification est assurée par l'émission par le processus emetteur() de couples (entier , byte).
L'entier identifie le processus propriétaire du byte ce qui permet la ventilation.
L'écriture d'un couple (entier, byte) en un seul paquet est possible de par la nature du protocole PROTO auquel est soumis le canal trans.canal. Finalement lorsque le récepteur a lu toutes les données (la fin des messages est déterminée par le caractère ' :') les buffers sont successivement écrits à l'écran par le processus recepteur() .

```
-- ch8_3.occ
--
#USE "course.lib"

PROTOCOL PROTO IS INT ; BYTE  :

PROC main(CHAN BYTE  scr!)
  [2]CHAN BYTE out.mplx         :
  CHAN PROTO trans.canal        :
  PAR
    -- processus emetteur 0
    VAL []BYTE message0 IS "ceci est le message emis par p0 : " :
    INITIAL INT i IS 0  :
    SEQ
      WHILE message0[i] <> ':'
        SEQ
          out.mplx[0] ! message0[i]
```

```
      i := i +1
  out.mplx[0]! ':'

-- processus emetteur 1
VAL []BYTE message1 IS "ceci est le message emis par p1 : " :
INITIAL INT i IS 0 :
SEQ
  WHILE message1[i] <> ':'
    SEQ
      out.mplx[1] ! message1[i]
      i := i +1
  out.mplx[1]! ':'


-- le processus multiplexeur
BYTE buffer0  :
BYTE buffer1  :
BOOL b0, b1   :
SEQ
  b0, b1 := TRUE, TRUE
  WHILE (b0 OR b1)
    ALT
      out.mplx[0]? buffer0
        SEQ
          trans.canal ! 0 ; buffer0
          IF
            buffer0 = ':'
              b0 := FALSE
            TRUE
              SKIP

      out.mplx[1]? buffer1
        SEQ
          trans.canal ! 1 ; buffer1
          IF
            buffer1 = ':'
              b1 := FALSE
            TRUE
              SKIP

  -- le processus recepteur
[256]BYTE buffer0 :
[256]BYTE buffer1 :
BYTE b            :
INT  i            :
BOOL b0, b1       :
INT idx0, idx1    :
SEQ
  b0 , b1 := TRUE, TRUE
  idx0 , idx1 := 0, 0
  WHILE (b0 OR b1)
    SEQ
      trans.canal? i ; b
      IF
```

```
          i = 0
            SEQ
              buffer0[idx0]:= b
              idx0 := idx0 + 1
              IF
                b = ':'
                  b0 := FALSE
                TRUE
                  SKIP
          i = 1
            SEQ
              buffer1[idx1]:= b
              idx1 := idx1 + 1
              IF
                b = ':'
                  b1 := FALSE
                TRUE
                  SKIP
    out.string(buffer0,0,scr!)
    flush(scr!)
    scr !'*n'
    out.string(buffer1,0,scr!)
    flush(scr!)
    scr !'*n'
:
```

L'intégralité des messages envoyés par les deux processus d'indices 0 et 1 au processus 'emetteur' se fait en un seul paquet grace au protocole STR qui véhicule et la longueur du message et le message proprement dit.

8.1.3 Processus ALT prioritaire

Si le mot clé PRI précède ALT le processus alternatif est prioritaire. Dans ce cas les gardes prêtes ne sont plus considérées comme équivalentes du point de vue du choix de l'une d'entre elles. L'ordre de leur écriture détermine leur priorité. La première garde éécrite est la plus prioritaire et la dernière écrite est la moins prioritaire.
L'exemple du chien de garde illustre l'usage de la priorité dans les processus alternatifs.

Implémanter un chien de garde (watchdog)

Un chien de garde est un processus qui se déclenche si une action particulière ne s'est pas manifestée dans un délais fixé. Le processus qui est décrit ici attend des frappes au clavier .Cette attente est incluse dans un processus alternatif prioritaire qui examine en premier lieu si une touche est enfonçée . Si la touche est la lettre 'q' il stoppe. Si l'attente entre deux frappes consécutives est supérieure à 10 secondes le chien de garde intervient et stoppe. En particulier, si à l'exécution du programme aucune touche n'est frappée dans un délai de 10 secondes le programme stoppe. Le processus alternatif étant prioritaire la frappe au clavier est examinée en premier. S'il y a eu frappe l'écho du caractère frappé est affiché et le timer est lu, au temps de la frappe, dans now1 .S'il n'y a pas de frappe la garde liée au timer est lu dans now2. La différence dif entre now2 et now1 donne l'intervalle de temps écoulé depuis la dernière frappe. Si cette

différence excède 10 secondes le processus de scrutation du clavier (sortir de WHILE) s'achève .
NB On remarque que la garde liée au timer est toujours prête. Il ne faut donc pas qu'elle masque une action en provenance du clavier.

```
-- ch8_4.occ
--
 #USE "course.lib"

PROC main(CHAN BYTE kbd?, scr!, err!)
  VAL INT sec IS 1000000      :
  BYTE car                    :
  BOOL encore                 :
  TIMER clock                 :
  INT now1, now2,diff, delay  :
  SEQ
    delay := 10*sec
    encore := TRUE
    clock ? now1
    WHILE encore
      PRI ALT
        kbd ? car
          SEQ
            scr ! car
            flush(scr)    -- vide le tampon de sortie
            scr! ' '
            clock ? now1  -- reactualise au temps de la frappe
            IF
              car = 'q'
                SEQ
                  scr ! '*n'
                  encore := FALSE
              TRUE
                SKIP
        clock ? now2
          SEQ
            diff := now2-now1
            IF
              diff >= delay
                SEQ
                  out.string("*nDelai ecoule *n",0,scr)
                  encore := FALSE
              diff < delay
                SKIP
    out.string("BY ...*n",0, scr)
:
```

8.2 La gestion des évènements asynchrones

Dans beaucoup de situations on est confronté au problème qui consiste à gérer des informations sans être bloqué à leur attente. Un cas typique, tiré de la vie quotidienne, est celui de la gestion du courrier postal. Je vaque à mes occupations et de temps en temps je consulte ma boite aux lettres. S'il y a du courrier je le retire sinon je retourne à mes occupations.
Un ALT prioritaire associé à un garde SKIP prète permet de ne pas rester bloqué sur la lecture d'un canal et fournit une solution simple à ce problème.

```
CHAN  PROTO canal:
SEQ
  -- processus
  -- processus
  PRI ALT
    canal? info
      processus   -- traite l'information
    TRUE & SKIP
      SKIP
  -- processus
```

Le programme qui suit illustre le plus simplement possible cette situation.
Deux processus : client et server s'exécutent en parallèle.
Toutes les 10 secondes le serveur envoie le message au client : "voila du courrier ! !".
Toutes les deux secondes le client regarde s'il y a du courrier. S'il y en a il affiche le message émis par le serveur sinon il affiche un point '.' et retourne à ses occupations sans être bloqué.

```
--ch8_5.occupations
--
#USE "course.lib"

PROTOCOL STR IS INT::[]BYTE:

PROC main(CHAN BYTE  scr!)
  CHAN STR ptt.canal  :
  PAR
    -- processus  server
    --
    --  delivre les messages
    --  toutes les 10 secondes
    VAL []BYTE lettre IS "voila du courrier !! *n"  :
    VAL INT delai IS 10000000 :
    TIMER clock :
    INT now     :
    WHILE TRUE
      SEQ
        clock? now
        clock? AFTER now PLUS delai

        ptt.canal! (SIZE lettre)::lettre

    -- processus client
    --
    INT now         :
    TIMER clock     :
    INT len         :
    [30]BYTE buffer :
    VAL INT d.seconde IS 2000000:
    WHILE TRUE
      SEQ
        -- travaille
        --
        --  consulte la boite aux lettres toutes les
```

```
        --  deux secondes
        clock? now
        clock? AFTER now PLUS d.seconde
        --  nettoie le buffer
        SEQ i=0 FOR 30
          buffer[i] := 0
        --
        --  Il y a du courrier ?
        --
        PRI ALT
          ptt.canal?len::buffer
            SEQ i=0 FOR len      --  oui
              scr! buffer[i]
          TRUE & SKIP
            SEQ                  --  non
              scr! '.'
              scr! '*n'
:
```

Chapitre 9

Les processus nommés et les fonctions

9.1 Les processus nommés

En Occam il est possible de nommer un processus . Par exemple on peut renommer SKIP en déclarant le processus rien.faire() comme ci dessous :

```
PROC rien.faire()
  SKIP
:
```

Invoquer le processus rien.faire() équivaut à invoquer SKIP comme on peut le voir dans l'exemple suivant :

```
#USE "course.lib"

PROC rien.faire()
  SKIP
:

PROC main(CHAN BYTE scr!)
  SEQ
    rien.faire()
    out.string("rien.faire() ne fait rien *n",0, scr!)
:
```

L'exemple montre en outre que le processus nommé rien.faire() est déclaré avant d'être invoqué par main().
Le texte du processus déclarant le processus rien.faire() est équivalant au texte du processus :

```
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  SEQ
    SKIP
    out.string("rien.faire() ne fait rien *n",0, scr!)
:
```

L'exemple de rien.faire() est trivial et nommer un processus, comme en programmation séquentielle, sert à structurer le texte d'un processus complexe pour en améliorer la lisibilité mais aussi à réutiliser le texte d'un processus.
En particulier il est essentiel de pouvoir encapsuler le code et les ressources mémoire d'un processus intégré dans une architecture parallèle en le nommant et en mettant en évidence (par le biais d'une liste de paramètres) les ressources, notamment les canaux, qui le relie aux autres processus.

Dans l'exemple qui suit le processus nommé add.quatre() est un processus séquentiel qui reçoit une valeur, lui ajoute 4 et envoie la somme sur le canal réservé à l'écran ce qui provoque l'affichage du résultat. La déclaration de add.quatre() fait apparaitre deux paramètres, l'un VAL associé au passage d'une valeur, l''autre CHAN associé à un canal.

```
-- ch9_1.occ
--
 #USE "course.lib"

PROC add.quatre(VAL BYTE i, CHAN BYTE screen!)
  BYTE local  :
  SEQ
    local := i + 4
    screen ! local
:

PROC main(CHAN BYTE scr!)
  SEQ
    add.quatre('A', scr!)
    scr! '*n'
:
```

A l'exécution la lettre 'E' s'affiche à l'écran.
Ici encore, à l'exécution, le texte du processus main() est équivaut à :

```
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  SEQ
     -- texte de add.quatre() ou 'A' se substitue a i
    BYTE local  :
    SEQ
      local := 'A' + 4
      scr ! local
    scr! '*n'
:
```

Dans ce troisième exemple de processus nommé le processus add4.bis() , qui renomme une affectation, a pour paramètre un entier (INT i). Ce type de paramètre est l'analogue du passage par référence des langages comme Pascal et permet, à l'exécution, d'accéder à une variable.

```
-- ch9_2.occ
--
#USE "course.lib"

PROC add4.bis(INT i)
  i := i+4
:

PROC main(CHAN BYTE scr!)
  INT x :
  SEQ
    x := 4
    add4.bis(x)
    out.int(x,0,scr!)
    scr! '*n'
:
```

A l'exécution 8 s'affiche à l'écran mettant en évidence que l'entier x, local à main() a été incrémenté de 4 .
A l'exécution le texte du processus main() est équivalent au texte qui suit :

```
 #USE "course.lib"

PROC main(CHAN BYTE scr!)
  INT x :
  SEQ
    x := 4
    -- texte de add4bis() ou x se substitue a i
    x := x+4
    out.int(x,0,scr!)
    scr! '*n'
:
```

Pour bien mettre en évidence les effets de bord associés aux passages par référence occam-π recommande de faire précéder ce type de paramètre de **RESULT**. Par exemple une écriture recommandée de add4.bis() est :

```
 PROC add4.bis(RESULT INT i)
   i := i+4
:
```

Il est des situations où un processus est amené à modifier son environnement et dans ce cas les paramètres passés par référence s'imposent. C'est du reste le cas de nombreuses fonctions système du module "course.lib" de Kroc comme par exemple :

```
PROC in.int (INT n, VAL INT max, CHAN BYTE in?, out!)

PROC in.string ([]BYTE s, INT length, VAL INT max, CHAN BYTE in?, out!)
```

Il n'est pas dans le style d'occam d'user du passage de paramètres par référence et nous le déconseillons chaque fois que cela est possible. Bien souvent le recours à une fonction (voir ci après) permet de l'éviter.
Par exemple add4bis() est avantageusement remplaçé par :

```
-- ch9_3.occ
--
#USE "course.lib"

INT FUNCTION  add4.ter(VAL INT i)
  INT aux:
  VALOF
    aux := i+4
    RESULT aux
:

PROC main(CHAN BYTE scr!)
  INT x :
  SEQ
    x := 4
    x := add4.ter(x)
    out.int(x,0,scr!)
    scr! '*n'
:
```

9.1.1 La déclaration d'un processus nommé

Syntaxe :

```
process.nomme        => {?,REC} PROC process.id ({0, parametres.formels})
                                {general}
                                {ressource}
                                 proc.compose
                         :

proc.compose         => proc.primitif | proc.standart

process.id           => m.identificateur
parametres.formels   => value.param | ref.param | chan.param

value.param          => VAL    data.type {1,m.dentificateur}
ref.param            => RESULT data.type {1,m.dentificateur}

chan.param           => channel.type   {1, chan.dir.id}
                     => []channel.type {1, chan.dir.id}

chan.dir.id          => canal.id? | canal.id!
```

Comme il a été vu au chapitre 1 les déclarations 'general' ne portent que sur des aspects structurels :
déclarations de types, de protocoles, de constantes, de processus nommés, de fonctions, de types de canaux mobiles, de types de processus mobiles.

Les déclarations 'ressource' concernent les ressources locales au processus :
déclarations de variables, de canaux, de timer, de barrières, de processus ou de canal mobile.
Une ressource, dans un processus nommé, est attachée à un processus standart. **Le squelette de la déclaration d'un processus nommé**.

En pratique,le processus encapsulé dans un processus nommé est soit un processus primitif, soit un processus composé. Dans le squelette qui suit, pour fixer les idées, nous avons choisi un processus séquentiel. (N'oublions pas que CSP, qui est le formalisme sur lequel Occam2.1 s'appuie, signifie Communicating Sequential Processes).

```
PROC process.id( liste de parametres formels)
  --
  declarations structurelles locales au processus :
  --
  declarations de ressources  locales au processus :
  --
  SEQ
    processus
    --
    processus
:
```

9.1.2 Référencer un processus nommé

Une fois déclaré un processus nommé est referençé par son identificateur suivi par une liste de ressources concordant ,dans l'ordre, en nombre et en types avec celles de la liste de l'en tête de la déclaration.
Exemple :

```
-- ch9_4.occ
--
PROC m.fonction(VAL INT u, RESULT INT x, y, CHAN INT in?)
  INT aux    :
  SEQ
    in? aux
    x := aux+5
    y := u+6
:

PROC main(CHAN BYTE scr!)
  CHAN INT canal      :
  INITIAL INT v IS 2  :
  INT a, b            :
  SEQ
    PAR
      canal! 4
      m.fonction(v, a, b, canal?) -- reference m.fonction()
    scr ! #30 + (BYTE a)
    scr ! '*n'
    scr ! #30 + (BYTE b)
    scr ! '*n'
:
```

A l'exécution de ce programme les chiffres 9 et 8 s'affichent montrant que les variables a et b ont été modifiées. En fait, à l'exécution, l'écriture m.fonction(v,a,b,canal) est équivalente au texte :

```
  INT aux:
  SEQ
    canal? aux
    a := aux+5
    b := 2+6
```

Syntaxe :

```
proc.instance     => process.id(0, actual.param)
actual.param      => expression | canal.param
```

9.1.3 Un programme de trie simple

Ce programme de tri exploite la technique du pipe line.

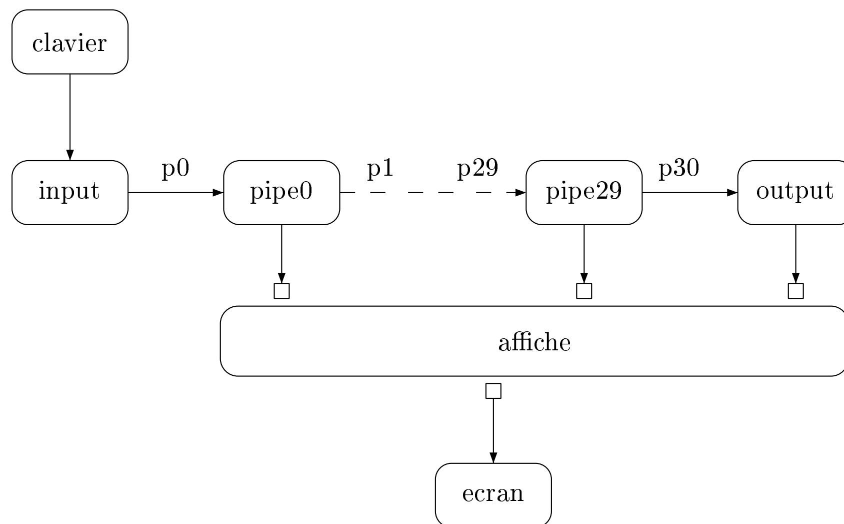


FIGURE 9.1 – Un programme de tri simple

```
-- ch9_5.occ
--
#USE "course.lib"

PROC affiche([]CHAN BYTE c, CHAN BYTE visu!)
  BYTE car :
  BOOL encore :
  SEQ
```

```
    encore := TRUE
    WHILE encore
      ALT i= 0 FOR 31
        c[i]? car
          IF
            car <> #0A
              SEQ
                cursor.x.y(10+(BYTE i),10, visu! )
                visu ! car
            car = #0A
              encore := FALSE
:

PROC  output(CHAN BYTE in.pipe?, out.visu!)    --  evite le DEADLOCK en lisant le
  BYTE x               :                       --  dernier maillon du pipe line
  SEQ
    in.pipe ? x
    WHILE x <> #0A
      SEQ
        out.visu ! x
        in.pipe ? x
    out.visu ! #0A
:

PROC  input(CHAN BYTE keyboard?, out.pipe!)
  BYTE  x       :
  SEQ
    keyboard ?  x
    WHILE x <> #0A
      SEQ
        out.pipe  ! x
        keyboard ? x
    out.pipe ! #0A
:

PROC  pipe(CHAN BYTE in.pipe?,out.pipe!, out.visu! )
  BYTE car.actuel ,car.lu , propage    :
  SEQ
    car.actuel := '0'
    in.pipe ? car.lu
    WHILE car.lu <> #0A    --  code de terminaison propage jusqu'au process output
      SEQ
        IF
          car.lu <  car.actuel
            propage := car.lu
          car.lu >= car.actuel
            SEQ
              propage    := car.actuel
              car.actuel := car.lu
        --
        PAR
          out.pipe ! propage
          out.visu ! car.actuel
        in.pipe  ? car.lu
```

```
    --
    out.pipe ! #0A
:

PROC main(CHAN BYTE clavier?, ecran! )
  [31]CHAN BYTE pipe.canal     :
  [31]CHAN BYTE visu.canal     :
  SEQ
    erase.screen(ecran!)
    cursor.x.y(1,1,ecran!)
    --
    out.string("  Pipe Line effectuant un trie*n",0, ecran!)
    out.string("  Frapper dans un ordre quelconque des caracteres*n",0, ecran!)
    out.string("  qui seront tries suivant la valeur de leur code ascii*n",0,ecran!)
    out.string("  Appuyer sur Return pour stopper le Pipe Line .*n*n",0, ecran!)
    --
    PAR
      input(clavier?, pipe.canal[0]!)
      output(pipe.canal[30]?,visu.canal[30]!)
      --
      PAR i= 0 FOR 30
        pipe( pipe.canal[i]? ,pipe.canal[i+1]! ,visu.canal[i] !)
      --
      affiche(visu.canal? , ecran!)
    out.string("*n*nBY ...*n",0,ecran)
:
```

Une chaine de processus s'exécute en parallèle reliés les uns aux autres par un canal d'entrée et un canal de sortie. Le processus input() injecte des données lues au clavier. Chaque processus pipe() lit son canal d'entrée et compare la valeur lue à la valeur courante stockée en interne.
Si le caractère lu est inférieur á la valeur courante il est écrit à son voisin de droite sur son canal de sortie sinon la valeur courante est propagée et le caractère lu prend sa place.
Le dernier processus output() ne fait que lire son entrée à partir du canal de sortie du dernier élément de la chaine .Il permet d'éliminer un risque d'interblocage en lisant le code de terminaison lié à la lecture de ENTER (code #0A) par input() et qui se propage sur toute la chaine.Il envoie alors ce code à affiche() qui se termine.
Notons que lorsqu'un processus pipe() de la chaine lit le code #0A il ne l'écrit pas à affiche() mais l'écrit à son voisin de droite et se termine.

9.1.4 Les cinq philosophes revisités (alting)

Le probléme des cinq philosophes est un grand classique résolu depuis longtemps par Djikstra à l'aide des sémaphores. Bien que trés puissants les sémaphores ne sont pas compatibles avec une architecture distribuée car ce sont des objets globaux, visibles par des processus concurrents.
Rappelons briévement le probléme :
Cinq philosophes occupent leur existence à penser (T think), manger (E eat) , dormir (S sleep) et, destin cruel , mourir (D dead) .
Ils peuvent manger à une table pourvue de cinq assiettes et de cinq fourchettes . Une fourchette est disposée entre deux assiettes .
Au centre de la table se trouve un plat de spagettis qui est, comme chacun sait ,la

nourriture par excellence des philosophes .
Les philosophes ne sont pas moins hommes et sujets à des caprices .
Pour manger confortablement leurs spagettis , lorsqu'ils ont faim , il leur faut deux fourchettes . Ils choisissent une assiette et prennent naturellement les fourchettes qui, se trouvent à droite et à gauche de celle ci .
On voit que pour qu'il n'y ait pas de conflit pour la possession des fourchettes seulement deux philosophes peuvent accéder à la table et manger à un instant donné.
Si un philosophe veut manger alors que les deux places sont prises il lui faudra faire la queue comme au restaurant et attendre qu'une place se libére .
Nous noterons (* suspendu) l'état d'un philosophe qui fait la queue.
Les états possibles pour un philosophe sont résumés par le graphe suivant :

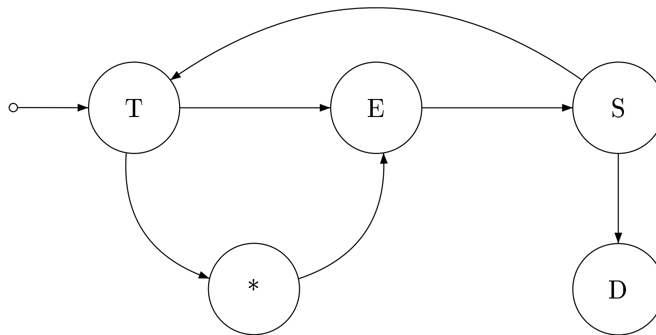


FIGURE 9.2 – Les états d'un philosophe

La gestion des places disponibles et celle de la file d'attente , pour satisfaire les besoins des philosophes qui ont faim , est faite par le processus manager() .
Ce dernier connait le nombre de places disponibles . Lorsqu'un philosophe désire manger il en fait la requête au manager . S'il y a une place de libre ce dernier le lui fait savoir et notre philosophe se met a table . S'il n'y a pas de place disponible le philosophe est mis en attente dans une fifo .Lorsqu'une place se libère et qu'un philosophe se trouve en tête de la fifo il en est informé par le manager et alors il peut manger .
Le programme ci dessous donne une implèmentation de cette analyse .

```
-- ch9_6.occ
--
#USE "course.lib"

PROC manager([]CHAN BYTE resp! , []CHAN BYTE req? , CHAN BYTE visu!)

  PROC print.status(VAL []BYTE status, CHAN BYTE out.visu!)
    SEQ
      SEQ i=0 FOR 5
        SEQ
          cursor.x.y(15 + (5*(BYTE i)),3, out.visu!)
          out.ch(status[i],0, out.visu!)
      flush(out.visu!)
  :

  INITIAL [3]INT fifo IS [i=0 FOR 3 | -1]     :
  INITIAL INT compte IS 0                      :
  INITIAL INT nb.places IS 2                  :
  INITIAL BOOL encore IS TRUE                 :
  INITIAL [5]BYTE status IS [i=0 FOR 5 | 'T'] :
  VAL INT vacant IS -1                        :
  BYTE request                                :
  INT head, tail                              :
  SEQ
    head , tail := 0, 0
    WHILE encore
      PRI ALT id=0 FOR 5
        req[id]? request
          CASE request
            'S'
              -- traite Sleep
              SEQ
                nb.places := nb.places + 1
                status[id] := 'S'

                -- remet la fifo en ordre
                -- active eventuellement un philosophe en attente
                INT philo :
                IF
                  fifo[head] >= 0
                    SEQ
                      nb.places := nb.places -1
                      philo := fifo[head]
                      fifo[head] := vacant
                      head := (head+1) REM 3
                      status[philo]:= 'E'
                      resp[philo]! 'E'
                  fifo[head] = vacant
                    SKIP
                print.status(status,visu!)

            'E'
              -- traite Eat
              IF
```

```
                -- il existe au moins une places disponible
                nb.places > 0
                  SEQ
                    resp[id]!'E'
                    nb.places := nb.places - 1
                    status[id] := 'E'
                    print.status(status, visu!)

                -- pas de places disponibles
                nb.places = 0
                  SEQ
                    status[id] := '**'
                    print.status(status,visu!)
                    fifo[tail] := id
                    tail := (tail +1) REM 3

            'T'
                -- il pense
              SEQ
                status[id] := 'T'
                print.status(status,visu!)

            'D'
              --  decede
              SEQ
                compte := compte + 1
                status[id] := 'D'
                print.status(status,visu!)
                IF
                  compte < 5
                    SKIP
                  compte = 5
                    encore := FALSE
:

PROC philosof(VAL INT id, CHAN BYTE resp.in?, CHAN BYTE req.out!)
  VAL INT sec IS 999900 :
  TIMER clock           :
  INT time              :
  BYTE ok               :
  INITIAL BOOL vivant IS TRUE   :
  INITIAL INT compte IS 0       :
  WHILE vivant
    SEQ
      req.out ! 'T'
      --  pense
      clock? time
      clock? AFTER time + ((id+1)*sec )
      --  desire manger
      req.out! 'E'
      resp.in? ok
      --  mange
      clock? time
      clock? AFTER time + (5*sec)
```

```
      -- want to Sleep
      req.out! 'S'
      clock? time
      clock? AFTER time + ((id+1)*sec)
      --  un jour de plus
      compte := compte+1
      IF
        compte < 10
          SKIP
        TRUE
          SEQ
            --  Dead
            req.out! 'D'
            vivant := FALSE
:

PROC main(CHAN BYTE scr!)
  [5]CHAN BYTE request  :
  [5]CHAN BYTE respond  :
  SEQ
    --
    -- preparation de l'affichage
    --
    erase.screen(scr!)
    cursor.x.y(0,0, scr!)
    out.string("philosophe : ",0,scr!)
    SEQ i=0 FOR 5
      SEQ
        cursor.x.y(15 + (5*(BYTE i)),0, scr!)
        out.ch(('0'+ (BYTE i)),0, scr!)
    --
    cursor.x.y(0,3, scr!)
    out.string("status     : ",0, scr!)
    SEQ i=0 FOR 5
      SEQ
        cursor.x.y(15 + (5*(BYTE i)),3, scr!)
        out.ch('T',0, scr!)
    cursor.invisible(scr!)
    --
    --  Fin de la preparation de l'affichage
    --
    PAR
      PAR i=0 FOR 5
        philosof(i, respond[i]?, request[i]!)
      manager(respond!, request?, scr!)
    out.string("*n*nBY ... *n*n",0, scr!)
    cursor.visible(scr!)
:
```

Un instantané typique de l'exécution du programme donne :

```
philosophe :  0    1    2    3    4

status     :  *    S    E    E    *
```

– le philosophe 0 est suspendu
– le philosophe 1 dort
– le philosophe 2 mange
– le philosophe 3 mange
– le philosophe 4 est suspendu

Analyse du programme

Coté philosophes :

Les philosophes ont des durées de vie différentes bien qu'exécutant chacun 10 cycles avant de s'achever
Le philosophe 0 s'achéve en premier et le philosophe 4 en dernier . Le processus manager() tient le compte du nombre de philosophes dont les cycles sont achevés . Lorsque les philosophes ont achevés leurs 10 cycles le processus manager() s'achéve à son tour ce qui assure une terminaison propre du programme .
Les philosophes passent, dans l'ordre, de l'états T vers l'état E puis S qui achéve un cycle . Le temps passé dans un état donné est variable et est fixé par le timer en fonction du numéro d'ordre du philosophe . La constante sec vaut à peu prés une seconde .Pour fixer les idées le philosophe 0 pense pendant 1 seconde et la durée d'un repas est de 5 secondes pour tous les philosophes.
Pour tout état , avant de passer dans un nouvel état, le philosophe en informe manager() en exécutant req.out ! 'code.etat'. En particulier lorsqu'un philosophe veut manger il exécute req.out ! 'E' et , dans ce seul cas , attend une réponse de manager() en exécutant resp.in, ? ok .
Si une place est disponible la réponse est immédiate et le philosophe passe dans l'etat 'E' sinon il est suspendu en lecture de resp.in ? ok et passe dans l'etat '*' .
Il sort de cet état par complétion de la lecture de resp.in ? ok .

Coté manager

Les requetes des philosophes sont gérées par un processus alternatif prioritaire .
Chaque requete est associée à un état qui est consigné dans [5]status pour y être affiché par print.status() .
Les seules requetes intéressantes sont 'E' (veut manger) et 'S' (sortir de table et dormir) .
Leur gestion s'appuie sur la variable nb.places qui donne le nombre de places disponibles au moment d'une requete et [3]fifo qui est une fifo circulaire a laquelle sont associes deux pointeurs : head (tete) et tail (queue). La fifo est initialisée à la valeur de vacant qui est -1 .head et tail sont initialisés en paralléle à zéro .
Le numéro d'un philosophe suspendu est consigné dans fifo[tail] . Le numéro d'un philosophe suspendu , autorisé a manger lorsqu'une place est liberée est consigné dans fifo[head]

traitement de 'E' :
Le numéro id du demandeur est le même que celui du canal lu par le processus alternatif. S'il y a une place libre status[id] est mis à 'E'et affiché . nb.places est décrementé , et l'exécution de resp[id] ! 'E' permet au philosophe de numero id qui a fait la requête de manger .
S'il n'y a pas de place libre id est consigné dans fifo[tail] et tail est incrementé modulo 3 (il ne peut pas y avoir plus de 3 processus suspendus). L'état status[id] est mis '*' et affiché. Le philosophe demandeur de numero id est suspendu puisque resp[id] ! 'E' n'est pas éxecuté

traitement de 'S' :

nb.places est incrémenté et status[id] mis a 'S' et affiché.
Si fifo[head] est >= 0 cela signifie qu'un philosophe de numéro fifo[head] est suspendu en lecture d'une réponse à sa demande de manger .Ce numéro est pris en compte dans la variable philo et fifo[head] est mis vacant . head est incrementé modulo 3 .Finalement l'exécution de resp[philo] ! 'E' débloque le philosophe de numéro philo . Son état status[philo] passe a 'E' puisqu'il peut se mettre a table et est affiché. Dans ce dernier cas la variable nb.places est de nouveau incrementée.
Si fifo[head] vaut vacant alors aucun philosophe n'est en attente et aucune action (SKIP) n'est entreprise.

traitement de 'T'
L'état du processus id est mis a 'T' : ie status[id] := 'T' et est affiché .

traitement de 'D'
Le philosophe informe manager() par cette information qu'il a accompli ses 10 cycles et son etat est mis a 'D' et affiché.
La variable compte , initialement à zéro ,est incrémentée .
Lorsque compte vaut 5 la variable encore est mise FALSE et le processus manager() s'achéve .

9.2 Les processus nommés récursifs

Curieusement occam-π n'autorise la récursion que pour les processus nommés et ne l'autorise pas pour les fonctions (définies ci dessous). La syntaxe des processus nommés récursifs est en tout point la même que pour les processus nommé sauf que leur déclaration est précédée de **REC**. Dans le chapitre consacré aux applications graphiques on trouvera un exemples important de processus nommés récursifs dans le programme de traçer de la courbe de Hilbert.
L'exemple classique de la fonction d'Ackermann est donné ci dessous :

```
-- ch9_7.occ
--
#USE "course.lib"

REC PROC ackermann(VAL INT m,n , RESULT INT result)
  INT p :
  IF
    m = 0
      result := n+1
    n = 0
      ackermann(m-1,1,result)
    (m > 0 ) AND (n > 0)
      SEQ
        ackermann(m, n-1, p)
        ackermann(m-1, p , result)
:

PROC main(CHAN BYTE scr!)
  INT m, n ,x   :
  SEQ
    m := 3
    n := 4
```

```
    x := 0
    ackermann(m,n,x)
    out.string("ackermann(3,4) : ",0,scr)
    out.int(x,0,scr!)
    scr! #0A
:
```

A l'exécution s'affiche ackermann(3,4) : 125

Un parcourt d'arbre :

Dans cet exemple un arbre est implémanté comme un tableau dont les éléments sont des structures. Les champs gauche et droite sont associés aux successeurs d'un sommet.

```
-- ch9_8.occ
--
DATA TYPE FEUILLE
  RECORD
    BYTE info        :
    INT gauche, droit:
:

DATA TYPE ARBRE IS [10]FEUILLE :


REC PROC parcourir(VAL ARBRE t,VAL INT i,  CHAN BYTE screen!)
  SEQ
    screen! t[i][info]
    screen! ' '
    SEQ
      IF
        t[i][gauche] > 0
          parcourir(t,t[i][gauche], screen)
        TRUE
          SKIP
      IF
        t[i][droit] > 0
          parcourir(t,t[i][droit], screen)
        TRUE
          SKIP
:


PROC main(CHAN BYTE scr!)
  VAL INT nill IS 0 :
  INITIAL ARBRE tree IS [ ['a',1,2],['z',3,4],['e',nill,5],['s',nill,nill],
                          ['n',8,nill],['m',6,7],['t',nill,9],
                          ['y',nill,nill],['f',nill,nill],['l',nill,nill] ]:

  SEQ
    parcourir(tree,0, scr!)
    scr! '*n'
:
```

9.3 Les fonctions

1 Une fonction est un processus nommé de type séquentiel (SEQ ,IF, WHILE, CASE).
2 Une fonction ne peut pas s'exécuter en parallèle avec un processus.
3 Les paramètres formels associes à sa déclaration sont des valeurs (VAL) .
4 Une fois terminée une fonction retourne au moins une valeur.

Le programme qui suit fait appel à la fonction somme() qui retourne la somme des éléments d'un tableau dont les valeurs sont passées en paramètre.

```
-- ch9_9.occ
--

#USE "course.lib"

INT FUNCTION somme(VAL []INT table)
  INT sum :
  VALOF
    SEQ
      sum := 0
      SEQ i=0 FOR SIZE table
        sum := sum + table[i]
    RESULT sum
:

PROC main(CHAN BYTE scr!)
  [10]INT tableau :
  INT resultat    :
  SEQ
    SEQ i=0 FOR 10
      tableau[i] := i
    resultat := somme(tableau)
    out.int(resultat,0,scr!)
    scr! '*n'
:
```

Ce programme utilise le processus out.int() qui fait partie de la librairie course et affiche un entier , en l'occurence 45 , somme des entiers de 1 à 9 .Il est impératif , si le fichier texte de ce programme est f1.occ , de le compiler : **kroc f1.occ -lcourse**

Si le texte de la fonction est réduit à une expression une forme courte d'écriture est possible comme le montre l'exemple suivant :

```
-- ch9_10.occ
--
#USE "course.lib"

INT FUNCTION is.odd(VAL INT i) IS (i /\ #1) :

PROC main(CHAN BYTE scr!)
  INT x , result  :
  SEQ
    x := 4
```

```
    result := is.odd(x)
    IF
      result = 0
        SEQ
          out.string("*n 4 est pair ",0, scr)
      TRUE
        SKIP
    x := 5
    result := is.odd(x)
    IF
      result = 1
        SEQ
          out.string("*n 5 est impair ",0, scr)
      TRUE
        SKIP
    out.string("*n BY... *n",0,scr)
:
```

La fonction is.odd() renvoie 1 si la valeur fournie est **impaire** et 0 sinon .On sait qu'un entier est impair si, dans son code binaire , le bit zéro vaut 1 .
La syntaxe de la déclaration des fonctions est curieuse en introduisant VALOF dont on ne comprend pas très bien la nécessité.Pour ce qui est de RESULT il renvoie au 'return' classique.

Syntaxe :

```
fonction           => fonction.normale | fonction.courte
fonction.normale   => {1, type.primitif} FUNCTION function.id ({0, parametre})
                        VALOF
                          processus
                          RESULT {1,expression}
                      :
fonction.courte    => {1, type.primitif} FUNCTION nom ({0, parametre})IS expression :
function.id        => m.identificateur
```

Si les résultats retournés sont multiples ils doivent se conformer, en ordre et en type , avec la liste associée à la déclaration. On trouvera pour terminer dans l'exemple qui suit une fonction qui retourne deux valeurs :

```
-- ch9_11.occ
--
#USE "course.lib"

INT, BOOL FUNCTION find.car(VAL BYTE char, VAL []BYTE string)
  BOOL ok    :
  INT ptr     :
  VALOF
    IF
      IF i=0 FOR SIZE string
        string[i]= char
          SEQ
            ok  := TRUE
            ptr := i
      TRUE
```

```
        SEQ
          ok  := FALSE
          ptr := -1
    RESULT ptr, ok
:

PROC main (CHAN BYTE scr!)
  INT position  :
  BOOL trouve   :
  VAL []BYTE message IS "azert/yuiop" :
  SEQ
    position , trouve := find.car('/' , message)
    IF
      trouve
        SEQ
          out.string("caractere trouve a la position : ",0,scr!)
          out.int(position,0, scr!)
      TRUE
        out.string(" caractere non trouve",0,scr!)
    out.string("*nBY ...*n",0,scr!)
:
```

A l'exécution s'affiche “caractere trouve a la position : 5” .

9.3.1 Référencer une fonction

Le référencement d'une fonction suit les mêmes règles que celles d'un processus nommé avec en plus les contraintes spécifiques aus paramètres des fonctions. Ces derniers sont obligatoirement associés à des passages par valeur.
Syntaxe :

```
fonc.instance        => fonc.id ({0, value.param})
```

Chapitre 10

Portée des déclarations

10.1 Les déclarations générales

Les déclarations générales décrivent **la nature** des ressources utilisées par un programme.
Ces ressources sont des données, des canaux, des processus nommés, des fonctions, des ressources dynamiques.
Les données sont décritent par leurs type, les canaux par leurs protocoles, les processus nommés et les fonctions par leur texte.
Les déclarations générales, au niveau zéro des indentations, sont visibles de toutes celles qui leur succèdent dans l'écriture du texte du programme.

Une déclaration générale située au même niveau d'indentation qu'un processus qui lui succède immediatement dans l'ordre de l'écriture n'est visible que de ce processus.

Syntaxe :

```
general             => data.struct | renommage
                    => protocole
                    => constante
                    => process.nomme | fonction
                    => mobil.type
```

10.2 Les déclarations de ressources

Liées à un processus. ces déclarations sont associées aux ressources locales (données, canaux, etc.) de ce dernier.
Elles sont uniquement visible du processus qui leur succède immédiatement dans l'ordre des écritures.

```
ressource           => variable | canal |timer
                    => barriere | mobile
```

Exemple :

```
-- ch10_1.occ
--
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  INT n    :                      -- 1  n est visible partout dans le processus SEQ
  SEQ                             -- 2
    n := 0                        -- 3
                                  -- 4  m est visible partout dans le processus WHILE
    INITIAL INT m IS 0   :
    WHILE n < 10                  -- 5
      SEQ                         -- 6
        n := n+1                  -- 7
        m := m + n                -- 8
        out.int(m,0, scr!)        -- 9   affiche n a l'ecran
        scr! '*n'
    scr! '*n'
    scr! '*n'
    out.int(n,0, scr!)            -- 10  affiche n a l'ecran
:
```

La portée de n, liée au processus séquentiel SEQ, s'étend des lignes 1 à 10 incluse.
La portée de m, liée au processus séquentiel WHILE, s'étend des lignes 5 à 9 incluse.
Autre exemple :

Dans cet exemple la variable i est declarée deux fois. La deuxième déclaration de i locale au deuxième processus séquentiel, masque la première.

```
-- ch10_2.occ
--
#USE "course.lib"

PROC main(CHAN BYTE scr!)
  INT i:
  --                                    Premier processus sequentiel
  SEQ                                   -- 1 portee du premier  i
    i := 0                              -- 2
    i := i + 10                         -- 3   i vaut 10

    --                                  Deuxieme processus sequentiel
    INT i:
    SEQ                                     -- 5 debut de la portee du second i
      i := 5                                -- 6 i vaut 5
      out.int(i,0, scr!)                    -- 7
      out.string("*n",0, scr!)              -- 8 affiche 5
      --   fin de la portee du second i
    out.int(i,0, scr!)                      -- 9   affiche 10
    out.string("*nBY ...",0, scr!)          -- 10
:
```

Bien que cet exemple ne soit pas un modèle de lisibilité il montre que la portée du premier entier i , attaché au premier processus sequentiel, est limitée aux lignes 1,2,3,9, 10 .Le deuxieme entier i est lié au deuxieme processus séquentiel et sa portée est associée

aux lignes 6,7,8.
Attention :

Une déclaration de variable (qui est une ressource) est liée à un processus et non à une déclaration de processus.
Par exemple le texte de “invalid.occ” ci dessous est invalide :

```
--
--  invalid.occ
--
INT i :
PROC main()
  SEQ
    i := 5
    SKIP
:
```

La compilation de “invalid.occ” donne naissance au message d’erreur ci dessous :

```
   1:INT i :
-----^
    :PROC main()
    :  SEQ
    :    i := 5
    :    SKIP
Error-occ21-invalid.occ(1)- item not allowed at outermost level of compilation unit
1 error found in source
```

10.2.1 Le cas particulier des processus parallèles

Les processus composant un processus parallèle ne peuvent pas partager de variable.
Considérons l’exemple suivant :

```
CHAN INT canal :
INT m.var      :
PAR
  -- pemier processus
  canal !  10
  -- deuxieme processus
  m.var := 20
  -- troisieme processus
  canal? m.var
```

Le compilateur détecte une référence a m.var au sein du deuxième et troisième processus. De toutes façons, la valeur finale de m.var serait indéterminée (10 ? ou 20 ?).

L’écriture suivante est admissible car la variable x n’est utilisée que par un seul processus.

```
PROC main(CHAN BYTE scr!)
  INT x:
  PAR
```

```
    x := 5
    INT y:
    y:= 4
:
```

L'écriture suivante du même programme est préférable.
Les processus composants un processus parallèle doivent déclarer leurs propres ressources localement dans l'esprit de CSP.

```
PROC main(CHAN BYTE scr!)
  PAR
    INT x:
    x := 5
    INT y:
    y:= 4
:
```

Si un processus nommé n'est référençé que par un seul processus il est préférable de le déclarer dans ce processus. Par exemple les deux textes suivants sont équivalents mais je conseille la seconde version :
Première version :
La déclaration de process1() étant au même niveau d'indentation que celle de process2() et la précédant alors process1() est visible dans process2().

```
PROC process1()
  --
  --
:

PROC process2()
  SEQ
    --
    --
    process1()
    --
:
```

Deuxième version :
La déclaration de process1() est au même niveau d'indentation que le processus séquentiel annonçé par SEQ et n'est visible que de ce processus séquentiel.

```
PROC process2()
  PROC process1()
    --
    --
  :
  SEQ
    --
    --
    process1()
    --
:
```

Chapitre 11

Les canaux partagés en occam-π

En occam-π les canaux peuvent être partagés, ce qui est impossible en occam2.1 car, comme on le sait, cette version d'occam ne permet qu'une liaison point à point sur un seul canal entre deux processus .
En occam-π on distingue 3 types de partage des canaux :

SHARED ! Un canal est partagé entre plusieurs écrivains et un lecteur.
SHARED ? Un canal est partagé entre plusieurs lecteurs et un écrivain.
SHARED Un canal est partagé entre plusieurs lecteurs et plusieurs écrivains.

Une déclaration de canal partagé est similaire a celle d'un canal ordinaire sauf que cette dernière doit être precédée de l'un des mots clés : SHARED ! , SHARED ? ou SHARED.

Syntaxe :

```
canal          => {?, shared} channel.type  {1, canal.id}:
shared         => SHARED | SHARED? | SHARED!
```

Exemples :

```
SHARED! CHAN INT  shared.write      :
SHARED? CHAN BYTE shared.read       :
SHARED  CHAN BYTE shared.read.write :
```

Dans ces exemples :
shared.write est déclaré comme un canal partagé en écriture seule .
shared.read est déclaré comme un canal partagé en lecture seule .
shared.read.write est déclaré comme un canal partagé en lecture et écriture.

11.1 Les blocs CLAIM

Lorsqu'un processus désire accéder à un canal partagé il le fait savoir en exécutant un bloc CLAIM.

Syntaxe :

```
bloc.claim              => CLAIM rw.channel.id
                               { processus }
rw.channel.id           => m.identificateur! | m.identificateur?
```

Exemple :

```
       --  le canal sh.write de protocole INT est partage en ecriture
       --  code est de type INT
      processus
      --
      CLAIM sh.write!
       -- entree du bloc CLAIM
        SEQ
          sh.write ! code
          processus 1
          ...
          processus n
      --  sortie du bloc CLAIM
      processus
```

Dans cet exemple lorsque le processus exécute CLAIM sh.write! il fait savoir qu'il désire accéder au canal sh.write partagé (en écriture dans l'exemple) .
S'il est seul a faire cette requête le canal sh.write est vérouillé à son profit et le processus (séquentiel dans l'exemple) qui suit CLAIM est exécuté .
Lorsque le processus associé au bloc CLAIM s'achéve la liaison avec le canal partagé est rompue .
Si au moins une requête d'accès au canal sh.write a été formulée par un autre processus écrivain antérieurement , alors le processus est suspendu et mis dans une file d'attente de type **fifo** . Il aura accès au canal lorsque, étant en tête de fifo , l'actuel processus détenteur de l'accés au canal aura achevé son propre bloc CLAIM .

11.2 Partage entre plusieurs écrivains et un lecteur

Etudions le programme suivant qui illustre , le plus simplement possible , le partage d'un canal entre plusieurs processus écrivains et un processus lecteur .

```
-- ch11_1.occ
--
PROC process(VAL INT id, SHARED CHAN INT c.serv!)
  -- ecrivain
  SEQ
    CLAIM c.serv!
      c.serv! id
:
```

```
PROC reader(CHAN INT c.serv?, VAL INT nb.proc, CHAN BYTE visu!)
  INT number  :
  SEQ i=0 FOR nb.proc
    SEQ
      c.serv? number
      visu! '0' + (BYTE number)
      visu! ' '
:

PROC main (CHAN BYTE scr!)
  SHARED! CHAN INT shared.chan :
  VAL INT NBPROC IS 5          :
  SEQ
    PAR
      reader(shared.chan?, NBPROC, scr!)
      PAR i=0 FOR NBPROC
        process(i, shared.chan!)
    scr! '*n'
:
```

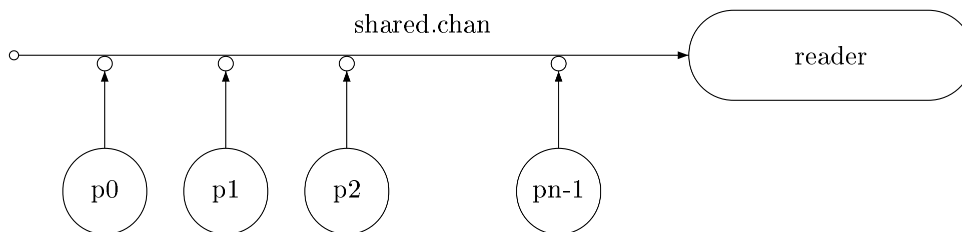


FIGURE 11.1 – Un canal partagé entre n ecrivains et 1 lecteur

Quelques remarques d'ordre syntaxique :

La déclaration PROC process(VAL INT id, SHARED CHAN INT c.serv!) dit :
le canal c.serv est partagé et l'écriture c.serv! est conforme à la convention générale en occam-π qui veut que c.serv! est un canal utilisé en écriture . **Par contre** bien que c.serv soit partagé en écriture ce fait **n'est pas mentionné dans SHARED**.
La déclaration PROC reader(CHAN INT c.serv?, VAL INT nb.proc, CHAN BYTE visu!) indique que c.serv est accédé en lecture mais **aucune mention n'est faite que c.serv est partagé.**. Le processus reader() se comporte , quand à sa relation avec c.serv comme avec un canal ordinaire .

Dans le bloc PAR du processus main() les références à shared.chan sont celles relatives à un canal ordinaire . Nulle mention n'est faite du caractère partagé de shared.chan tel que cela ressort de sa déclaration SHARED! CHAN INT shared.chan.

Une brève analyse de ce programme .

Sans préjuger de la manière dont le parallélisme est géré à son plus bas niveau par kroc

les processus process() que sont les écrivains font leur requête d'accès au canal partagé dans l'ordre de leur numéro (le i du PAR récupéré par chaque processus par id).Le premier à accéder à shared.chan est le processus 0 et les autres sont mis éventuellement en fifo . Une fois l'accès obtenu ils écrivent leur numéro qui est lu par reader() qui effectue séquentiellement autant de requétes que de processus écrivains .
Sur le terminal s'affiche : 0 1 2 3 4 .

11.3 Partage entre plusieurs lecteurs et un écrivain

Tout ce qui vient d'être dit sur le partage en écriture d'un canal entre plusieurs processus écrivains et un processus lecteur se transpose au cas du partage d'un canal en lecture entre de multiples processus lecteurs et un processus écrivain .

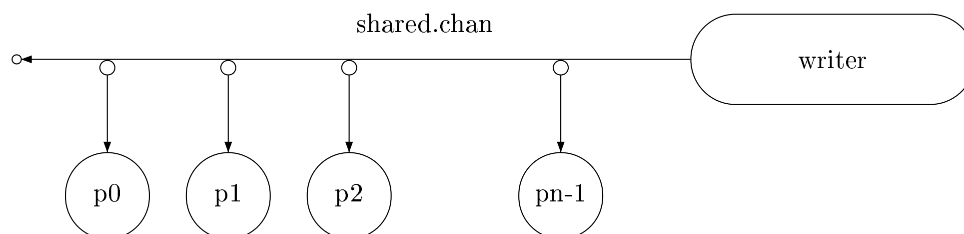


FIGURE 11.2 – Un canal partagé entre n lecteurs et 1 écrivain

Le programme ci dessous est un exemple simple de ce type de partage.

```
-- ch11_2.occ
--
PROC process(VAL INT id, SHARED CHAN INT c.serv?, CHAN BYTE out.visu! )
  -- processus lecteur
  INT i      :
  SEQ
    CLAIM c.serv?
      c.serv? i
    IF
      i = id
        out.visu ! '0' + (BYTE i)
      TRUE
        out.visu ! 'A' + (BYTE i)
:
```

```
PROC writer(CHAN INT c.serv!, VAL INT nb.proc)
  VAL [10]INT hazard IS[0,1,3,2,0,4,2,1,3,4]  :
  SEQ i=0 FOR nb.proc
    c.serv! hazard[i]
:

PROC affiche([]CHAN BYTE c.scr?,VAL INT nb.proc, CHAN BYTE ecran)
  INITIAL BOOL encore IS TRUE   :
  INITIAL INT compte IS 0       :
  BYTE car
  WHILE encore
    SEQ
      ALT i=0 FOR nb.proc
        c.scr[i]? car
          SEQ
            ecran ! car
            ecran ! ' '
      compte := compte+1
      IF
        compte < nb.proc
          SKIP
        TRUE
          encore := FALSE
:

PROC main (CHAN BYTE scr!)
  SHARED? CHAN INT shared.chan :
  [10]CHAN BYTE c.visu         :
  VAL INT NBPROC IS 5          :
  SEQ
    PAR
      affiche(c.visu?, NBPROC, scr!)
      writer(shared.chan!, NBPROC)
      PAR i=0 FOR NBPROC
        process(i, shared.chan?, c.visu[i])
    scr! '**'
    scr! '*n'
:
```

Les remarques syntaxiques faites précédemment s'appliquent encore ici mot pour mot sauf que shared.chan est déclaré SHARED ?.
SHARED est référençé dans PROC process(VAL INT id, SHARED CHAN INT c.serv ?, CHAN BYTE out.visu) qui rappelle que c.serv est partagé par les lecteurs et qu'il est accédé en lecture .
De même le processus writer() voit c.serv comme un canal ordinaire auquel il accède en écriture .
writer() écrit en séquence des entiers dans un ordre défini par le []hazard .
Lorsqu'un lecteur accède au canal partagé (remarquer l'écriture du bloc CLAIM en lecture) si l'entier lu est identique à son numéro il l'affiche (en écrivant dans c.visu[i])sinon il affiche le caractère qui suit la lettre 'A' du nombre reçu .
L'affichage s'interpréte facilement si l'on pense que les processus accédent au canal shared.chan dans l'ordre de leur numéro fourni par le PAR répliqué .
Sur le terminal s'affiche : 0 1 D C A associées aux valeurs 0,1,3,2,0 ecrites par writer()

Le processus 0 reçoit 0 affiche 0
Le processus 1 reçoit 1 affiche 1
Le processus 2 reçoit 3 affiche D (3eme caractère passé A)
ect ...

11.4 Partage entre plusieurs écrivains et plusieurs lecteurs

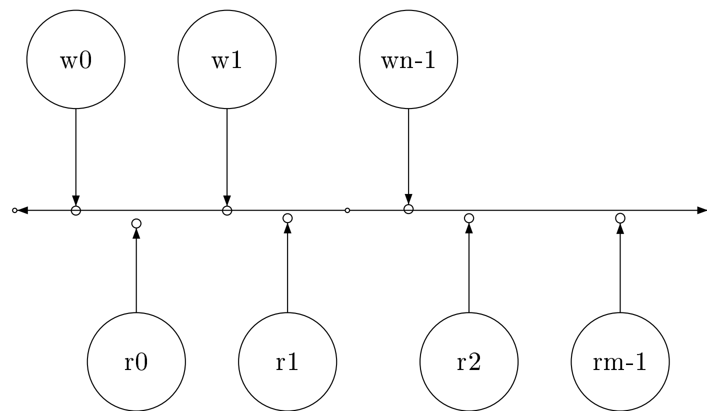


FIGURE 11.3 – Un canal partagé entre m lecteurs et n écrivains

```
ch11_3.occ
--
PROC write(VAL INT id, SHARED CHAN BYTE c.write!)
  IF
    id = 0
      CLAIM c.write!
        c.write! 'A'
    id = 3
      CLAIM c.write!
        c.write! 'C'
    TRUE
      SKIP
:
```

```
PROC read(VAL INT id, SHARED CHAN BYTE c.read?, SHARED CHAN BYTE out.visu!)
  BYTE car  :
  IF
    id = 1
      CLAIM c.read?
        SEQ
          c.read? car
          CLAIM out.visu!
            out.visu! car
    id = 7
      CLAIM c.read?
        SEQ
          c.read? car
          CLAIM out.visu!
            out.visu! car
    TRUE
      CLAIM out.visu!
        out.visu! '0' + (BYTE id)
:

PROC affiche(CHAN BYTE c.scr?,VAL INT nb.proc, CHAN BYTE ecran)
  INITIAL BOOL encore IS TRUE   :
  INITIAL INT compte IS 0       :
  BYTE car                      :

  WHILE encore
    SEQ
      ALT i=0 FOR nb.proc
        c.scr ? car
          SEQ
            ecran ! car
            ecran ! ' '
      compte := compte+1
      IF
        compte < nb.proc
          SKIP
        TRUE
          encore := FALSE
:

PROC main(CHAN BYTE scr!)
  SHARED CHAN BYTE shared.chan  :
  SHARED! CHAN BYTE c.visu      :
  VAL INT NPROC IS 10           :
  SEQ
    PAR
      affiche(c.visu?, NPROC, scr!)
      PAR i= 0 FOR NPROC
        PAR
          read(i, shared.chan? , c.visu!)
          write(i, shared.chan!)
    scr! '*n'
:
```

Sur le terminal s'affiche :

```
0 2 3 4 5 6 8 9 A C
```

interpretation

Toujours sans préjuger de la gestion du parallélisme par kroc on fait l'hypothèse (raisonnable) que les 10 lecteurs sont lançés en premier et les 10 écrivains en second .
Les lecteurs 0,2,3,4,5,6,8,9 ne demandent pas d'accès au canal partagé shared.chan et affichent leur numéro .
Les lecteurs 1 et 7 font une demande d'accès à shared.chan .
Les ecrivains n'étant pas encore lancés ils sont suspendus dans une fifo .
Les écrivains sont maintenant lançés .
Seuls les ecrivains 0 et 3 font une demande d'ecriture . L'écrivain 0 exécute CLAIM et écrit A qui est lu par le lecteur 1 en tête de fifo . A s'affiche . L'écrivain 3 exécute CLAIM . Dans un premier temps il est suspendu puis lorsque le CLAIM du processus 0 s'achéve il écrit C qui est lu par le lecteur 7 qui l'affiche .
On notera que dans ce programme l'affichage se fait par le canal c.visu partagé en écriture par les lecteurs .

remarque importante

Lorsque deux processus l'un ecrivain , l'autre lecteur , communiquent à travers un canal partagé en lecture et en écriture nous verrons (cf le chapitre 13) que la bonne méthode consiste a créer un canal mobile entre ces deux processus une fois qu'ils se sont mis d'accord pour partager des informations .

11.4.1 Les cinq philosophes revisités (shared)

On trouvera ici une nouvelle écriture du programme des cinq philosophes étudié dans le cadre des processus alternatifs. Les deux tableaux de canaux du premier programme sont avantageusement remplacés par seulement deux canaux. L'un est partagé en écriture pour les requètes et l'autre en lecture pour l'acquitement des requètes.

```
-- ch11_4.occ
--
#USE "course.lib"

PROTOCOL REQUEST IS INT; BYTE :

PROC manager(CHAN BYTE resp!, CHAN REQUEST req?, CHAN BYTE visu!)

  PROC print.status(VAL []BYTE status, CHAN BYTE out.visu!)
    SEQ
      SEQ i=0 FOR 5
        SEQ
          cursor.x.y(15 + (5*(BYTE i)),3, out.visu!)
          out.ch(status[i],0, out.visu!)
      flush(out.visu!)
  :
  BYTE request                            :
  INT head, tail, id                      :
```

```
INITIAL [3]INT fifo IS [i=0 FOR 3 | -1]     :
INITIAL INT morts IS 0                      :
INITIAL INT nb.places IS 2                  :
INITIAL BOOL encore IS TRUE                 :
INITIAL [5]BYTE status IS [i=0 FOR 5 | 'T'] :
VAL INT vacant IS -1                        :
SEQ
  head , tail := 0, 0
  WHILE encore
    SEQ
      req? id; request
      CASE request
        --  ............................
        'S'
          -- traite Sleep
          SEQ
            nb.places := nb.places + 1
            status[id] := 'S'
            -- remet la fifo en ordre
            -- active eventuellement un philosophe en attente
            INT philo :
            IF
              fifo[head] >= 0
                SEQ
                  nb.places := nb.places -1
                  philo := fifo[head]
                  fifo[head] := vacant
                  head := (head+1) REM 3
                  status[philo]:= 'E'
                  resp! 'E'
              fifo[head] = vacant
                SKIP
            print.status(status,visu!)
          --  ............................
        'E'
         -- traite Eat
          IF
            -- il existe au moins une places disponible
            nb.places > 0
              SEQ
                resp!'M'
                nb.places := nb.places - 1
                status[id] := 'E'
                print.status(status, visu!)

            -- pas de places disponibles
            nb.places = 0
              SEQ
                status[id] := '**'
                print.status(status,visu!)
                fifo[tail] := id
                tail := (tail +1) REM 3
        --      ...........................
```

```
          'T'
            -- il pense
            SEQ
              status[id] := 'T'
              print.status(status,visu!)
          --  ................................
          'D'
            --  decede
            SEQ
              morts := morts + 1
              status[id] := 'D'
              print.status(status,visu!)
              IF
                morts < 5
                  SKIP
                morts = 5
                  encore := FALSE
:


PROC philosof(VAL INT id, SHARED CHAN BYTE resp.in?, SHARED CHAN REQUEST req.out!)
  VAL INT sec IS 999900 :
  TIMER clock           :
  INT time              :
  BYTE ok               :
  INITIAL BOOL vivant IS TRUE   :
  INITIAL INT compte IS 0       :
  WHILE vivant
    SEQ
      CLAIM req.out!
        req.out ! id; 'T'
      --  pense
      clock? time
      clock? AFTER time + ((id+1)*sec )
      --  desire manger
      CLAIM req.out!
        req.out! id; 'E'
      CLAIM resp.in?
        resp.in? ok
      --  mange
      clock? time
      clock? AFTER time + (5*sec)
      -- want to Sleep
      CLAIM req.out!
        req.out! id; 'S'
      clock? time
      clock? AFTER time + ((id+1)*sec)
      --  un jour de plus
      compte := compte+1
      IF
        compte < 10
          SKIP
        TRUE
          SEQ
```

```
            --  Dead
            CLAIM req.out!
              req.out! id; 'D'
            vivant := FALSE
:

PROC main(CHAN BYTE scr!)
  SHARED! CHAN REQUEST request  :
  SHARED? CHAN BYTE respond     :
  SEQ
    erase.screen(scr!)
    cursor.x.y(0,0, scr!)
    out.string("philosophe : ",0,scr!)
    SEQ i=0 FOR 5
      SEQ
        cursor.x.y(15 + (5*(BYTE i)),0, scr!)
        out.ch(('0'+ (BYTE i)),0, scr!)
    --
    cursor.x.y(0,3, scr!)
    out.string("status     : ",0, scr!)
    SEQ i=0 FOR 5
      SEQ
        cursor.x.y(15 + (5*(BYTE i)),3, scr!)
        out.ch('T',0, scr!)
    cursor.invisible(scr!)
    --
    --  Fin de la preparation de l'ecran
    --
    PAR
      PAR i=0 FOR 5
        philosof(i, respond?, request!)
      manager(respond!, request?, scr!)
    out.string("*n*nBY ... *n*n",0, scr!)
    cursor.visible(scr!)
:
```

Chapitre 12

La mobilité des données en occam-π

12.1 Le type mobile.data.type

Toute ressource de type data.type peut être déclarée mobile en occam-π .

Syntaxe :

```
mobile.data.type      => MOBILE data.type
```

Exemples :

```
-- on suppose le type data.struct MY.RECORD defini

MOBILE INT
MOBILE []BYTE
MOBILE MY.RECORD
```

12.2 La déclaration des variables mobiles

Une variable mobile est une variable dont le type est mobile.data.type. Elle se déclare comme une variable ordinaire. Sa déclaration crée une référence relative à la ressource et se comporte comme un pointeur non initialisé.
Exemple :

```
-- on suppose le type.struct MY.RECORD defini

MOBILE INT i1, i2           :
MOBILE []BYTE  mob.string    :
MOBILE MY.RECORD mob.record  :
```

12.2.1 La création des variables mobiles

L'allocation de ressource avec initialisation peut, comme pour une variable ordinaire, se faire par affectation depuis une variable de même type (mobile ou non) déjà définie soit par une affectation particulière qui utilise l'opérateur MOBILE. Cet opérateur alloue la ressource demandée sans l'initialiser.
Exemple :

```
-- ch12_1.occ
--
#INCLUDE "course.module"

PROC main(CHAN BYTE scr!)
  MOBILE []BYTE s1, s2, s3              :
  MOBILE INT mob.int                    :
  INITIAL [10]BYTE data IS "QSDFGHJKLM"     :
  INITIAL INT n IS 10                   :

  INITIAL MOBILE []INT mt IS MOBILE [20]INT :
  SEQ
    mob.int := 17                -- alloue et initialise mob.int

    s1 := MOBILE [n]BYTE         -- alloue  a s1 une zone de 10 BYTE
    SEQ i=0 FOR 10               -- initialise cette zone
      s1[i] := '0' + (BYTE i)

    s2 := "AZERTYUIOP"           -- alloue et initialise s2
    --
    s3 := data                   -- alloue et initialise s3
:
```

L'exemple 's1 := MOBILE [n]BYTE' est significatif d'une allocation dont la taille n'est connue qu'à l'exécution.

12.2.2 Perte de référence, clonage

En occam-π toute ressource est identifiée par un identificateur unique.
Il en découle qu'une variable mobile qui est assignée dans une affectation ou écrite dans un canal perd sa référence.
Exemple :

```
PROC main(CHAN BYTE scr!)
  MOBILE []BYTE s1, s2    :

  SEQ
    s1 := "AZERTYUIOP"           -- alloue et initialise s1
    s2 := s1                     -- s1 devient indefinie
:
```

Dans l'exemple suivant on montre une perte de référence suite à une écriture canal .
Les canaux déclarés CHAN MOBILE véhiculent des références.

```
-- ch12_2.occ
--
#USE "course.lib"

PROC recoit(CHAN MOBILE []BYTE data.in? , CHAN BYTE ecran!)
  MOBILE []BYTE s :
  SEQ
    data.in ? s
    out.string(s , 0, ecran )
:

PROC emet(CHAN MOBILE []BYTE data.out!)
  MOBILE []BYTE s , t :
  SEQ
    s := "AZERTYUIOP*n"
    data.out! s   --  perte de la reference de s
    t := s        -- WARNING s undefined here
:

PROC main(CHAN BYTE scr!)
  CHAN MOBILE []BYTE message :
  PAR
    emet(message!)
    recoit(message? , scr!)
:
```

Dans le processus emet() une fois exécuté "data.out ! s" la valeur de s est a nouveau indéterminée et l'affectation "t := s" déclenche un warning de la part de kroc à la compilation.

Le clonage :

L'opérateur de clonage CLONE permet de **copier** une ressource référençée par une variable sans déférencer cette dernière.
Exemple :

```
-- ch12_3.occ
--
#INCLUDE "course.module"

PROC main(CHAN BYTE scr!)
  MOBILE []BYTE s1, s2 :
  SEQ
    s1 := "azerty *n"
    s2 := CLONE s1           -- s1 n'est pas dereferencee
    out.string(s1,0, scr!)
    out.string(s1,0, scr!)
:

    A l'execution s'affiche :
    azerty
    azerty
```

Pour terminer nous laissons le lecteur étudier le programme suivant (issus des master's classes de l'Université de Kent fourni avec la documentation de Kroc).

```
-- ch12_4.occ
--
 #USE "course.lib"

PROC producer (CHAN MOBILE []BYTE c!)
  MOBILE []BYTE s:

  WHILE TRUE
    SEQ i = 1 FOR 26
      SEQ
        s := MOBILE [i]BYTE
        SEQ j = 0 FOR i
          s[j] := BYTE ('a' + j)
        c ! s
:

PROC consumer (CHAN MOBILE []BYTE c?, CHAN BYTE visu!)
  MOBILE []BYTE s:

  WHILE TRUE
    SEQ
      c ? s
      out.string (s, 0, visu!)
      visu ! '*n'
:

PROC main (CHAN BYTE screen!)
  CHAN MOBILE []BYTE c :
  PAR
    producer (c!)
    consumer (c?, screen!)
:
```

12.3 Une barrière partielle

Les barrières standart d'occam-π permettent à un nombre déterminé de processus enrolés sur ces dernières de se synchroniser. Une barrière partielle permet à un sous ensemble quelconque de processus parallèles de se synchroniser sans qu'il soit fait usage d'une barrière mais seulement d'une utilisation judicieuse de canaux partagés en écriture.

L'étude qui suit est inspirée d'un article de Peter Welch sur le problème dit 'Santa Claus Problem' qui peut être consulté à : `http://santaclausproblem.cs.unlv.edu/occpi.html`
Le programme :

```
-- ch12_5.occ
--
#USE "course.lib"

PROC p.bariere(VAL INT n, CHAN BYTE ask?, remove?)
  BYTE any  :
  WHILE TRUE
```

```
    SEQ
      SEQ i=0 FOR n
        ask? any
      --
      --
      SEQ i=0 FOR n
        remove? any
:

PROC go.to.work(VAL INT n, SHARED CHAN BYTE  ecran!, CHAN BYTE c.work?)
  MOBILE []BYTE request  :
  SEQ
    request := MOBILE [n]BYTE
    WHILE TRUE
      SEQ
        SEQ i=0 FOR n
          c.work? request[i]
        CLAIM ecran!
          SEQ
            flush(ecran!)
            out.string("*nrendez vous de : ",0, ecran!)
            SEQ i=0 FOR n
              ecran! 'A' + request[i]
:

PROC process(VAL INT i, SHARED CHAN BYTE ask.bar!,
                        remove.bar!, SHARED CHAN BYTE p.work!)
  VAL INT sec IS 1000000  :
  VAL BYTE any IS 1       :
  TIMER clock             :
  INT now                 :
  WHILE TRUE
    SEQ
      clock? now
      clock? AFTER now PLUS (i*sec)
      SEQ
        CLAIM ask.bar!     -- s'enrole sur p.barier
          ask.bar ! any
        CLAIM p.work!      -- requete a go.to.work()
          p.work! (BYTE i)
        clock? now
        clock? AFTER now PLUS (3*sec)
        --
        CLAIM remove.bar!   -- relache p.barier
          remove.bar ! any
:

PROC main(SHARED CHAN BYTE scr!)
  SHARED! CHAN BYTE ask.bar, remove.bar:
  SHARED! CHAN BYTE ask.work  :
  PAR
    PAR i=0 FOR 18
      process(i, ask.bar!, remove.bar!, ask.work!)
    p.barier(3, ask.bar?, remove.bar?)
```

```
    go.to.work(3, scr!, ask.work?)
:
```

Dès que 3 processus parmis 18 ont pu faire une requète à go.to.work() ce dernier affiche “rendez vous de : XX YY ZZ”. Au préalable ces processus s'enrolent sur p.barier() et dès que le nombre 3 requis est atteint ils accèdent à go.to.work(). Après un certain délai ils relachent p.barier().
On remarquera le role important joué par les canaux partagés en écriture et on rappelle que les processus qui accèdent un canal partagé sont suspendus dans un ordre géré par une fifo s'ils n'ont pas obtenu satisfaction.

Chapitre 13

La mobilité des canaux en occam-π

En occam2.1 les canaux sont fixes et déterminés une fois pour toutes à la compilation. Ils sont associés à la déclaration de graphes d'interconnection de processus au sein d'un réseau.
Les canaux mobiles d'occam-π , créés à l'exécution, sont adaptés à la gestion de liaisons dynamiques entre processus.

13.1 La déclaration CHAN TYPE

Un canal mobile, constitué d'un faisceau de canaux ordinaires, est caractérisé par ses deux bouts.
La liaison de deux processus à travers un canal mobile peut s'effectuer dès lors que chacun des processus est en possession de l'un des bouts du canal mobile.
Chaque canal du faisceau de canaux est identifié par un identificateur directionnel et un protocole.
L'ensemble des caractéristiques (identificateur + protocole) des canaux constitutifs d'un canal mobile fait l'objet de la déclaration de structure CHAN TYPE.

L'identificateur 'mobile.chan.struct.id' définit le type de tout canal mobile dont la structure est conforme à celle déclarée par CHAN TYPE.

Syntaxe :

```
mobile.channel.type        => CHAN TYPE mobile.chan.struct.id
                                MOBILE RECORD
                                  {1,mobil.chan.field}
                              :
mobil.chan.field           => CHAN protocole.id chan.dir.id
mobile.chan.struct.id      => M.identificateur
chan.dir.id                => m.identificateur? | m.identificateur!
```

Exemple :

```
 CHAN TYPE M.CANAL
   MOBILE RECORD
     CHAN BYTE  req? :
     CHAN BYTE resp! :
:
```

M.CANAL est le type de tout canal mobile composé de deux canaux identifiés par les champs req ?, resp !, chacun de ces canaux étant de protocole BYTE . Bien que les

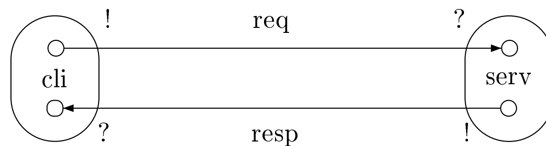


FIGURE 13.1 – Un canal mobile de type M.CANAL

deux bouts d'un canal mobile soient associés à la même structure définie par CHAN TYPE l'un des bouts est distingué comme bout client et l'autre comme bout serveur afin de pouvoir préciser le sens des transits des informations dans ce canal.

Convention :

Par convention le bout serveur d'un canal mobile voit le faisceau de canaux dont les directions en lecture et écriture sont celles définies par la déclaration CHAN TYPE.

13.1.1 La déclaration d'un canal mobile

La déclaration d'un canal mobile est définie par celle de ses deux bouts.
L'indicateur directionnel dit qui est le bout serveur et qui est le bout client.

Exemple de la déclaration d'un canal mobile de bouts 'serv' et 'cli' de type M.CANAL.

```
M.CANAL? serv :   -- declare le bout serveur du canal mobile  de type M.CANAL
M.CANAL! cli  :   -- declare le bout client  du canal mobile  de type M.CANAL
```

13.1.2 La création d'un canal mobile

Comme pour toutes les ressources de type mobile sa création suit le shéma :

```
ressource.mobile.type ressource.id :            -- declaration
ressource.id := MOBILE ressource.item           -- creation
```

Dans le cas des canaux mobiles la ressource est identifiée par le couple formé par ses deux bouts de type mobile.chan.struct.id.

```
bout.client , bout.serveur := MOBILE mobile.chan.struct.id
```

Exemple :

```
cli , serv  := MOBILE M.CANAL  -- cree le canal mobile de type M.CANAL
                               -- et dont les bouts sont serv et cli
```

13.1.3 Les canaux constitutifs d'un canal mobile

Les bouts d'un canal mobile sont des structures particulières dont les champs donnent l'accès aux canaux constitutifs de ce canal.
Par exemple le couple (cli, serv) définit un canal mobile à deux canaux identifiés par les champs req et resp. Plus précisemment (cli[req] ! , serv[req] ?) définit un premier canal en lecture sur serv et en écriture sur cli.
Le couple (cli[resp] ? , serv[resp] !) définit un second canal en lecture sur cli et en écriture sur serv.
On aura noté que la convention qui veut que le bout déclaré serveur voit les canaux constitutifs du canal mobile en accord avec la déclaration CHAN TYPE est respectée.

Exemple :

```
-- ch13_1.occ
--
CHAN TYPE M.CANAL
  MOBILE RECORD
    CHAN BYTE req?  :
    CHAN BYTE resp! :
:

PROC main(CHAN BYTE scr!)
  M.CANAL? serv  :                -- declaration du canal mobile
  M.CANAL! cli   :                -- par celle de ses deux bouts
  SEQ
    serv, cli := MOBILE M.CANAL   -- creation du canal mobile
    PAR
      -- processus A
      BYTE b.client :
      SEQ
        cli[req]! 'A'
        cli[resp]? b.client
        scr! b.client
        scr! '*n'

      processus B
      BYTE b.server :
      SEQ
        serv[req]?  b.server
        serv[resp]! b.server +1
:
```

Le processus A émet 'A' sur le canal (cli[req] !, serv[req] ?) en direction du processus B. Ce dernier incrémente la valeur reçue de 1 et la retourne sur le canal (serv[resp] !, cli[resp] ?) en direction du processus A qui affiche la valeur reçue : 'B'.

13.2 La mise en oeuvre de la mobilité

En Occam les processus ne communiquent qu'à travers des canaux.
Dans le cas classique en occam2.1 : Les processus communiquent par un canal ordinaire. L'un lit dans le canal l'autre écrit dans ce canal.

Dans le cas mobile en occam-π : L'un des processus posséde un bout d'un canal mobile et l'autre processus posséde l'autre bout de ce canal mobile.

Dans l'exemple précédent les processus A et B exploitent directement les bouts du canal mobile dès sa création.
Cet exemple, hormi le fait qu'il permet de comprendre comment utiliser les canaux constitutifs d'un canal mobile, est de peu d'intéret.

Des processus nommés qui désirent communiquer à travers un canal mobile ont deux solution :
1 - Chaque processus acquière le bout desiré par le biais d'un passage de paramètre par référence.
2 - Chaque processus acquière le bout desiré par la lecture de ce dernier dans un canal (ordinaire ou mobile).

13.2.1 Passage des bouts par référence

L'exemple ci dessous reprend le programme précédent où, cette fois, les processus A et B sont remplacés par des processus nommés process.a() et process.b(). Une référence à cli est passée à process.a() et une référence à serv est passée à process.b().
Exemple :

```
-- ch13_2.occ
--
CHAN TYPE M.CANAL
  MOBILE RECORD
    CHAN BYTE req?  :
    CHAN BYTE resp! :
:

PROC process.a(M.CANAL! canal, CHAN BYTE out!)
  BYTE reponse  :
  SEQ
    canal[req]! 'A'                -- emet une requete
    canal[resp]? reponse
    out! reponse
    out! '*n'
:

PROC process.a(M.CANAL! canal, CHAN BYTE out!)
  BYTE requete  :
  SEQ
    canal[req]? requete           -- attend une requete
    canal[resp]!requete + 1       -- retourne la reponse
:
```

```
PROC main(CHAN BYTE scr!)
  M.CANAL? serv  :
  M.CANAL! cli   :
  SEQ
    serv, cli := MOBILE M.CANAL  -- creation
    PAR
      process.a(cli , scr!)      -- passage des bouts par reference
      process.b(serv )
:
```

On aura noté que le paramètre canal de process.a(M.CANAL ! canal, CHAN BYTE out !) est associé au type M.CANAL ! qui est le type de cli dont la référence est passée à ce processus. Même remarque pour le paramètre canal de process.b(M.CANAL ? canal) qui est le type de serv passé par référence à process.b().

13.2.2 Passage des bouts par lecture d'un canal

Un canal permet le transfert de toute ressource qui satisfait aux protocoles reconnus par le langage.
En occam-π le type mobile.channel.type est un protocole valide.

Syntaxe :

```
protocole            => protocole.fixe
                     => protocole.variable
                     => protocole.mobile

protocole.mobile     => mobile.channel.type
```

Dans l'exemple qui suit on reprend le thème des deux exemples précédents sauf que cette fois le bout cli du canal est passé à process.a() par le canal canal.cli de protocole M.CANAL ! et le bout serv à process.b() par le canal canal.serv de protocole M.CANAL ?.

```
-- ch13_3.occ
--
#INCLUDE "course.module"


CHAN TYPE M.CANAL
  MOBILE RECORD
    CHAN BYTE req?  :
    CHAN BYTE resp! :
:

PROC process.a(CHAN M.CANAL! canal.s, CHAN BYTE ecran!)
  BYTE reponse    :
  M.CANAL! canal  :
  SEQ
    canal.s ? canal        -- lit le bout client

    canal[req]! 'A'         -- emet une requete
    canal[resp]? reponse
    ecran! reponse
    flush(ecran!)
    ecran! '*n'
:

PROC process.b(CHAN M.CANAL? canal.c)
  BYTE requete    :
  M.CANAL? canal  :
  SEQ
    canal.c? canal            -- lit le bout serveur

    canal[req]? requete       -- attend une requete
    canal[resp]!requete + 1   -- retourne la reponse
:

PROC main(CHAN BYTE scr!)
  CHAN M.CANAL! canal.cli   :
  CHAN M.CANAL? canal.serv  :
  M.CANAL? serv  :
  M.CANAL! cli   :
  SEQ
    serv, cli := MOBILE M.CANAL
    PAR
      canal.cli! cli
      process.a(canal.cli? , scr!)

      canal.serv! serv
      process.b(canal.serv? )
:
```

13.2.3 Un serveur simple

L'exemple qui suit, qui est une légère modification du programme a3.occ des Master's Class de Adam Sampson de l'Universite de Kent, est fourni avec la distribution de Kroc.
Il montre comment des liens dynamiques peuvent s'établir entre un serveur et des clients qui désirent se connecter à lui.

Des clients désirent obtenir un nombre aléatoire d' un serveur.
Lorsqu'un client veut faire la requète d'un nombre il établit, dans un premier temps, une liaison mobile entre lui et le serveur.
Pour ce faire il crée un canal mobile dont il transfère le bout serveur 'svr' au serveur en utilisant le canal 'register' partagé en écriture entre tous les clients.
Lorsque qu'un client fait la requète d'un nombre proprement dite il la fait en utilisant le canal mobile crée à l'étape précédente. Le serveur tire un nombre au hazard et l'écrit au client en utilisant le canal mobile qui le lie à ce dernier.
Il y a, coté serveur, autant de bouts de type serveur que de clients puisque chaque client crée son propre canal mobile.
Le serveur, lorsqu'il reçoit un bout, stocke ce dernier dans le tableau 'ends' déclaré MOBILE.

Un tableau de bouts doit être déclaré MOBILE

Le programme :

```
-- ch13_4.occ
--
#INCLUDE "course.module"

PROTOCOL DICE.REQ
  CASE
    roll
    done
:

CHAN TYPE DICE.CT
  MOBILE RECORD
    CHAN DICE.REQ req?  :
    CHAN INT resp!      :
:


PROC server (CHAN DICE.CT? request.cli?)
  VAL INT MAX.CLIENTS IS 5      :

  INITIAL INT seed IS 2509197   :          -- utilise par random()
  INITIAL INT compte IS 0       :
  INITIAL BOOL encore IS TRUE   :

  INITIAL[MAX.CLIENTS]BOOL in.use IS [i = 0 FOR MAX.CLIENTS | FALSE] :
  MOBILE [MAX.CLIENTS] DICE.CT? ends    :
  DICE.CT? new.end                      :
  WHILE encore
    PRI ALT
```

```
      request.cli ? new.end                -- les requetes de creation de canal mobile
        IF i = 0 FOR MAX.CLIENTS           -- sont prioritaires
          NOT in.use[i]
            SEQ
              in.use[i] := TRUE            -- enregistre le bout
              ends[i] := new.end           -- marque ce dernier disponible

      ALT i = 0 FOR MAX.CLIENTS
        in.use[i] & ends[i][req]? CASE     -- une demande de nombre est faite
          roll
            INT n:
            SEQ
              n, seed := random( 6, seed)  -- cree le nombre aleatoire
              ends[i][resp] ! (n+1)        -- l'ecrit au client

          done                             -- demande de deconnection
            SEQ                            --
              in.use[i] := FALSE
              compte := compte+1
              IF
                compte < MAX.CLIENTS
                  SKIP
                TRUE                       -- le serveur se termine apres
                  encore := FALSE          -- MAX.CLIENTS demandes de nombres.

:

PROC client (VAL INT id, SHARED CHAN DICE.CT? send.server!, SHARED CHAN BYTE out!)
  DICE.CT! cli:
  DICE.CT? svr:
  SEQ
    cli, svr := MOBILE DICE.CT
    CLAIM send.server!             -- ecrit le bout svr au serveur
      send.server ! svr

    INT n :
    SEQ
      cli[req] ! roll              -- demande d'un nombre aleatoire
      cli[resp] ? n                -- reponse du serveur

      cli[req]! done               -- code de deconnection

      CLAIM out!
        SEQ
          out.string ("Client #", 0, out!)
          out.int (id, 0, out!)
          out.string (" rolled a ", 0, out!)
          out.int (n, 0, out!)
          out ! '*n'
:

PROC main (SHARED CHAN BYTE out!)
  SHARED! CHAN DICE.CT? register:
  PAR
```

```
    server (register?)
    PAR i = 0 FOR 5
      client (i, register!, out!)
:
```

13.2.4 Lectures et écritures multiples entre processus

L'exemple qui est traité ici généralise le précédent.

Si un écrivain veut correspondre avec un lecteur (ou l'inverse, les protocoles sont identiques) il crée une liaison mobile avec un serveur, comme dans l'exemple précédant, en utilisant un canal partagé en écriture.
Cette liaison l'identifie auprès du serveur comme ecrivain (resp lecteur) et comme demandeur d'une correspondance avec un lecteur donné (resp ecrivain donné).
Le serveur enregistre les requètes. Dès qu'un couple de requètes (ecrivain, lecteur) a été identifié par le serveur ce dernier crée un canal mobile dont les bouts sont envoyés aux deux correspondants à travers les liaisons mobiles initialement crées par ces derniers.
Cet exemple montre que l'on peut transférer les bouts d'un canal mobile en utilisant un canal également mobile comme l'illustre la déclaration CHAN TYPE LINK.

Le protocole :
Sans nuire à la généralité on suppose que le lecteur i fait en premier sa requète en faveur de l'ecrivain j .
Il commence par créer un canal mobile entre lui et le serveur , en écrivant le bout serveur de ce canal au processus serveur() en utilisant le canal 'register' partagé en écriture .
Il écrit ensuite au serveur, à travers le canal mobile ainsi crée, le code 1 qui dit qu'il est lecteur, son numéro i et celui j de son correspondant.

Lorsque le serveur reçoit cette requète, il l'enregistre et en particulier le bout coté serveur qui le relie au lecteur i.
Comme le lecteur i est le premier à vouloir correspondre il est suspendu en lecture de son canal mobile en attente de la réponse du serveur.
Lorsque l'écrivain j présente sa propre requète, identique en tout point a celle du lecteur sur le canal mobile qu'il a également crée (sauf le code 0 qui dit qui dit qu'il est écrivain), le serveur identifie cette requète à la requète faite précédemment par le lecteur i.

Le serveur possède maintenant les deux bouts serveur des deux canaux mobiles dont les bouts clients sont en la possession de i et de j.
Le serveur crée alors le canal canal mobile qui va relier **directement** i à j et envoie les bouts de ce canal via les deux canaux mobiles qui le relie à ces derniers.

Le programme :

```
-- ch13_5.occ
--
#INCLUDE  "course.module"

DATA TYPE REQ
```

```
  RECORD
    INT type        :       --  1 lecteur ; 0 ecrivain
    INT ici         :       --  identifie le demandeur
    INT autre       :       --  identifie le correspondant
:

CHAN TYPE M.PRT             -- structure du canal mobile
  MOBILE RECORD             -- joignant un lecteur a un ecrivain
    CHAN BYTE data!  :
:

CHAN TYPE LINK              --  structure du canal mobile liant
  MOBILE RECORD             -- les lecteurs et les ecrivains au
    CHAN REQ  request?  :   -- serveur
    CHAN M.PRT! link.c! :
    CHAN M.PRT? link.s! :
:

VAL INT LECTEUR  IS 1 :
VAL INT ECRIVAIN IS 0 :


PROC affiche(CHAN BYTE data.in? , CHAN BYTE visu!)
  BYTE car  :
  SEQ i=0 FOR 10
    SEQ
      data.in? car
      visu! car
      visu! ' '
      data.in? car
      visu! car
      visu! ':'
      flush(visu!)
:

PROC server(CHAN LINK? requ.in?)
  --
  --  fit.read.write[j] = i <=> l'ecrivain j veut
  --  correspondre avec le lecteur i
  --
  INITIAL [10]INT fit.read.write IS [i=0 FOR 10 | -1 ]:
  MOBILE []LINK? ends.read  :
  MOBILE []LINK? ends.write :
  LINK? link.end            :
  REQ requete               :
  INT id.read, id.write     :
  SEQ
    -- alloue l'espace pour recueillir les bouts
    -- des canaux mobiles le liant aux lecteurs et aux
    -- ecrivains
    ends.read  := MOBILE[10]LINK?
    ends.write := MOBILE[10]LINK?
    SEQ i=0 FOR 20
      SEQ
```

```
requ.in? link.end
link.end[request]? requete
IF
  requete[type] = 1
    --  la requete est celle d'un lecteur
    SEQ
      id.read := requete[ici]    -- id du lecteur
      id.write:= requete[autre]  -- id du correspondant

      --  enregistre le cote servr du canal mobile
      --  qui lie le serveur au lecteur
      ends.read[id.read] := link.end
      IF
        fit.read.write[id.write] >= 0
          -- un ecrivain a anterieurement manifeste
          -- de correspondre avec ce lecteur
          --
          -- le serveur cree les bouts du futur canal
          -- qui reliera lecteur et ecrivain
          M.PRT! cli   :
          M.PRT? serv  :
          SEQ
            cli, serv := MOBILE M.PRT
            -- envoie les bouts du canal mobile
            -- au lecteur et a l'ecrivain
            PAR
              ends.read[id.read][link.c]  ! cli
              ends.write[id.write][link.s]! serv
        TRUE
          --  aucun ecrivain ne s'est manifeste
          --  enregistre la requete du lecteur

          fit.read.write[id.write] := id.read

  requete[type] = 0
    -- requete d'un ecrivain
    -- sa gestion est en tout point identique
    -- a celle du lecteur
    SEQ
      id.read := requete[autre]
      id.write:= requete[ici]

      ends.write[id.write] := link.end

      IF
        fit.read.write[id.write] >= 0
          --
          M.PRT! cli   :
          M.PRT? serv  :
          SEQ
            cli, serv := MOBILE M.PRT
            PAR
              ends.read[id.read][link.c]  ! cli
              ends.write[id.write][link.s]! serv
```

```
                TRUE
                  --
                  fit.read.write[id.write] := id.read
:


PROC reader(VAL INT id, SHARED CHAN LINK? register.out!, SHARED CHAN BYTE visu!)
  INITIAL [10]INT table.write IS [5,0,4,3,7,8,2,9,1,6]:
  --
  -- table.write[i] = j <=> le lecteur i veut correspondre avec j
  REQ requete     :
  LINK? serv      :
  LINK! cli       :
  M.PRT! data.in  :
  BYTE info       :
  SEQ
    cli, serv := MOBILE LINK

    -- veut correspondre
    -- etablit un canal mobile avec le serveur
    -- en passant a ce dernier le bout serveur.
    CLAIM register.out!
      register.out ! serv
    --
    --  le canal mobile le liant au serveur est etabli
    --  le lecteur envoie sa requete en utilisant
    --  ce canal
    requete[type] := LECTEUR
    requete[autre]:= table.write[id]
    requete[ici] := id

    cli[request]! requete
    --
    --  se met en attente de la lecture du bout client
    --  du canal mobile qui le lie directement a l'ecrivain
    cli[link.c]? data.in
    --
    --  le lecteur lit le BYTE ecrit par l'ecrivain
    data.in[data]? info
    -- demande  l'affichage de  l'information recue
    -- au processus affiche()
    CLAIM visu!
      SEQ
        visu! '0' + (BYTE id)
        visu! info

:

PROC writer(VAL INT id, SHARED CHAN LINK? register.out!)
  INITIAL [10]INT table.read IS[1,8,6,3,2,0,9,4,5,7]:
  --
  -- table.read[i] = j <=> l'ecrivain i veut correspondre avec j
  --
  --  La gestion de l'ecrivain est pratiquement identique ,
```

```
  --  a celle du lecteur

  REQ requete     :
  LINK? serv      :
  LINK! cli       :
  M.PRT? data.out :

  SEQ
    cli, serv := MOBILE LINK

    CLAIM register.out!
      register.out ! serv
    --
    requete[type] := ECRIVAIN
    requete[autre] := table.read[id]
    requete[ici] := id

    cli[request]! requete
    --
    cli[link.s] ? data.out
    --   Ecrit son identificateur au lecteur
    data.out[data]! '0' + (BYTE id)
    --
:

PROC main (CHAN BYTE scr!)
  SHARED! CHAN LINK? register:
  SHARED! CHAN BYTE out.scr  :
  SEQ
    PAR
      affiche( out.scr? , scr!)
      server (register?)
      PAR i = 0 FOR 10
        PAR
          reader (i, register!, out.scr!)
          writer (i, register!)
    out.string("*nBY ...*n",0,scr!)
:
```

Chapitre 14

La mobilité des processus en occam-π

L'état d'un processus mobile est caractérisé par son **contexte** constitué des ressources auquelles il a accés, de son code, et d'un indicateur sur la prochaine action à exécuter. Lorsqu'un processus est suspendu, son contexte est sauvergardé et lui est restitué lorsque le processus est réactivé.

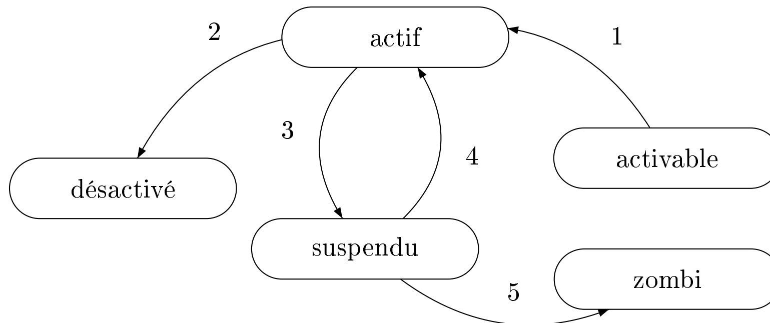


FIGURE 14.1 – Les états d'un processus mobile

- 1 le processus est activé par invocation de son identificateur .
  Il est mis dans la liste des processus actifs .Il exécute du code.
- 2 le processus , ayant exécuté tout son code, est désactivé.
  Il n'est plus reconnu en tant que processus ..
- 3 le processus exécute SUSPEND et est supprimé de la liste des processus actifs. Il est mis en attente dans une liste de processus activables .Il ne peut pas, dans cet état, exécuter du code .Son contexte est sauvegardé.
- 4 Par invocation de son identificateur le processus est remis dans la liste des processus actifs et peut de nouveau exécuter du code compte tenu de son contexte qui lui est restitué.

– 5 Le processus est rayé de la liste des processus actifs bien qu'il n'ai pas exécuté la totalité de son code. Cet état est conjecturé dans l'état actuel des références données sur les processus occam-π .

14.1 La déclaration PROC TYPE

La déclaration PROC TYPE associée à la signature d'un processus définit le type d'une nouvelle ressource : celle de processus mobile.
Les paramètres formels qui interviennent dans une signature de processus associée à la déclaration PROC TYPE sont soit des canaux, soit des barrières.

Syntaxe

```
mobil.proc.type    => PROC TYPE proc.type.id IS ( { 0, sync.param.id } ) :

sync.param.id      => chan.param | barriere.param
proctype.id        => M.identificateur
```

Exemple :

```
PROC TYPE PT IS (CHAN BYTE out!):
```

Une déclaration de processus nommé qui implémente 'proc.type.id' est nécessaire à la création d'une ressource de type 'MOBILE proc.type.id'.

```
mobil.proc.nomme  =>   MOBILE PROC process.id ({0, sync.param.id }) IMPLEMENTS proc.type.id
                         {general}
                         {ressource}
                         proc.compose
                      :
```

La signature 'process.id ({0, sync.param.id})' doit être identique à celle de la déclaration PROC TYPE.

Exemple :

La déclaration qui suit est associée à la création d'une ressource de type MOBILE PT.

```
PROC TYPE PT IS (CHAN BYTE out!):

MOBILE PROC print (CHAN BYTE screen!) IMPLEMENTS PT
  INITIAL [10]BYTE data  IS "AZERTYUIO*n" :
  SEQ i=0 FOR 10
    screen! data[i]
:
```

14.2 Les processus mobiles

Un processus mobile est une ressource dont le type est 'MOBILE proc.type.id'.

Par exemple :

```
MOBILE PT mobil.process :      -- declare mobil.process
```

Déclare 'mobil.process' comme identifiant un processus mobile de type MOBILE PT.

Le processus mobile 'mobil.process' est rendu **activable** en exécutant :

```
mobil.process := MOBILE print  -- mobil.process est activable
```

On peut dire qu'à ce stade mobil.process() se comporte comme un alias de print().
Il peut maintenant être exécuté comme un processus normal. L'exemple qui suit résume cette introduction aux processus mobiles .

```
-- ch14_1.occ
--
PROC TYPE PT IS (CHAN BYTE out!):

MOBILE PROC print (CHAN BYTE screen!) IMPLEMENTS PT
  INITIAL [10]BYTE data  IS "AZERTYUIO*n" :
  SEQ i=0 FOR 10
    screen! data[i]
:

PROC main(CHAN BYTE scr!)
  MOBILE PT mobil.process :           -- declare mobil.process
  SEQ
    mobil.process := MOBILE print     -- mobil.process est activable
    mobil.process(scr!)               -- mobil.process est actif
    --                                -- mobil.process est desactive
:
```

L'exécution de ce programme affiche AZERTYIO à l'écran puis saute une ligne avant de rendre la main .

14.3 La suspension d'un processus mobile

Un processus mobile se suspend **volontairement** en exécutant SUSPEND.
Son contexte est sauvegardé et lui est restitué à la prochaine activation.
Exemple :

```
-- ch14_2.occ
--
PROC TYPE PT IS (CHAN BYTE out!):

MOBILE PROC mobproc(CHAN BYTE visu!) IMPLEMENTS PT
  INITIAL BYTE b IS 'A' :
  INT i :
  SEQ
    i := 0
    SUSPEND
    b := b + (BYTE i) --  i=0
    visu ! b          --  b='A'
```

```
    visu ! '*n'
    i := i+1           --  i=1
    SUSPEND
    b := b + (BYTE i) --  i=1
    visu ! b           --  b='B'
    visu ! '*n'
    i:= i+1            --  i=2
    SUSPEND
    b := b + (BYTE i) --  i=2
    visu ! b           --  b='D'
    visu ! '*n'
:


PROC main(CHAN BYTE scr!)
  MOBILE PT pm :
  SEQ
    pm := MOBILE mobproc
    SEQ i=0 FOR 4
      pm(scr!)
:
```

Ce programme montre que la reprise, après chaque suspension, respecte et les valeurs des variables avant suspension et le code qui se déroule dans le bon ordre.
En général un processus mobile se suspend en interaction avec son environnement par le biais des canaux . Considérons le programme suivant associé à une **mauvaise** gestion de SUSPEND .

```
-- ch14_3.occ
--
PROC TYPE PT IS (CHAN INT in?, CHAN BYTE out!):

MOBILE PROC print(CHAN INT in.c?, CHAN BYTE screen!) IMPLEMENTS PT
  INITIAL INT init IS 0 :
  INT code             :
  SEQ
    SEQ i= init FOR 10
      screen! 'A' + (BYTE i)
    screen ! '*n'
    init := 10
    in.c? code
    IF
      code = 0
        SEQ
          SEQ i= init FOR 10
            screen! 'A' +(BYTE i)
          screen! '*n'
      code > 0
        SEQ
          init := 15
          SUSPEND
          in.c? code
          SEQ i = init FOR 10
            screen! 'A' + (BYTE i)
```

```
        screen! '*n'
:


PROC main(CHAN BYTE scr!)
  MOBILE PT mob.proc  :
  CHAN INT canal      :
  SEQ
    mob.proc := MOBILE print
    SEQ
      PAR
        canal! 1 -- v0
        mob.proc(canal?, scr!)
      scr! '**'
      scr! '*n'
      PAR
        canal! 0 -- v1
        mob.proc(canal? , scr!)
:
```

Plusieurs configurations des valeurs v0 , v1 écrites dans canal vont être examinées.
Dans la configuration actuelle v0 vaut 1 et v1 vaut 0.
Sur l'écran s'affiche :

```
ABCDEFGHIJ
*
PQRSTUVWXY
```

Dans cette configration mob.proc affiche "ABCDEFGHIJ" puis lit le code 1 . La variable init vaut 15 et mob.proc est suspendu . Cette suspension entraine la fin du premier PAR .Une étoile s'affiche . Le deuxième PAR s'exécute et mob.proc est activé avec son contexte restauré (init vaut 15)ce qui entraine l'affichage de PQRSTUVWXY". On notera que dans ce cas de figure la lecture de la valeur 0 lue par mob.proc sur le canal n'a aucune incidence.(elle permet seulement d'éviter l'interblocage) .

Soit maintenant la configuration où v0 vaut 0 et v1 la valeur 1 .
Sur le terminal s'affiche :

```
ABCDEFGHIJ
KLMNOPQRST
*
KRoC: application error, stopped.
```

En effet les deux lignes "ABCDEFGHIJ" et "KLMNOPQRST" correspondent à l'affichage généré par mob.proc lorsque le code 0 est lu . **MAIS** alors mob.proc a fini d'exécuter son code et redevient **inactif** . N'étant plus reconnu le deuxième PAR ne peut pas s'exécuter d'où le message lançé par kroc .
Exercice : Que doit on faire pour que mob.proc soit reconnu au deuxiéme PAR ?.

Ci dessous on trouvera une variante de print() qui assure une terminaison propre quelque soient les valeurs de v0 et de v1 .Sur le terminal est affiché :

```
xxxxxxxxxx
*
yyyyyyyyyy
```

x prend la valeur de v0 (0 ou 1) .y prend la valeur de v1 (0 ou 1).

```
MOBILE PROC print(CHAN INT in.c?, CHAN BYTE screen!) IMPLEMENTS PT
  INITIAL INT init IS 0 :
  INT code             :
  INITIAL BOOL encore IS TRUE:
  WHILE encore
    SEQ
      in.c? code
      IF
        code = 0
          SEQ
            SEQ i= 0 FOR 10
              screen! '0'
            screen! '*n'
            init := init+1
            IF
              init = 2
                encore := FALSE
              TRUE
                SUSPEND
        code = 1
          SEQ
            SEQ i= 0 FOR 10
              screen! '1'
            screen! '*n'
            init := init+1
            IF
              init = 2
                encore := FALSE
              TRUE
                SUSPEND
:
```

14.4 La mobilité d'un processus dans un réseau

Nous conservons les déclarations faites dans les exemples précédents .
Le type de données défini par MOBILE PT permet de considérer le contexte d'un processus mobile de signature définie par PT comme une donnée ordinaire et donc comme une donnée transférable dans un canal.

```
MOBILE PT mob.proc      :  -- declare un processus mobile de signature PT
CHAN MOBILE PT mp.chan  :  -- declare un canal apte a transferer le contexte de mob.proc
```

Dans ces déclarations mp.chan est un canal **ordinaire** (non mobile!) par lequel peut transiter le contexte d'un processus mobile dont la signature est définie par PT .
Comme il a été dit un processus mobile ne peut être transféré que si son état est soit activable soit suspendu . Considérons l'exemple suivant :

```
-- ch14_4.occ
--
PROC TYPE PT IS (CHAN BYTE out!)   :
```

```
MOBILE PROC print(CHAN BYTE out!) IMPLEMENTS PT
  SEQ
    SEQ i=0 FOR 10
      SEQ
        out! 'A'
    out ! '*n'
    SUSPEND
    SEQ i=0 FOR 10
      SEQ
        out! 'B'
    out ! '*n'
:

PROC proc(CHAN MOBILE PT cpt?, CHAN BYTE visu!)
  MOBILE PT mobil.proc  : -- container pour mobil.proc
  SEQ
    SEQ i=0 FOR 10
      visu! '**'
    visu! '*n'
    cpt? mobil.proc       -- identifie par mobil.proc , mob.proc est importe
    mobil.proc(visu!)     -- mob.proc est execute
:

PROC main(CHAN BYTE scr!)
  MOBILE PT mob.proc  :
  CHAN MOBILE PT mbc  :
  SEQ
    mob.proc := MOBILE print
    mob.proc(scr!)      -- mob.proc est actif
    PAR
      mbc ! mob.proc    -- exporte mob.proc sur mbc
      proc(mbc?, scr!)  -- proc() importe mob.proc sur mbc
:
```

A l'exécution de ce programme s'affiche sur le terminal :

```
AAAAAAAAAA
**********
BBBBBBBBBB
```

Une fois actif mob.proc exécute le code qui précède SUSPEND ce qui entraine l'affichage des "AAAAAAAAAA" suivi du saut de ligne .
Une fois mob.proc suspendu le processus parallèle est lançé .
proc() affiche "**********" suivi du saut de ligne puis se met en lecture du canal mbc passé en paramètre.
(Notons que le processus d'écriture dans mbc est suspendu tant que proc() n'a pas achevé ses écritures .)
Le transfert du contexte de mob.proc se fait et il est affecté à mobil.proc .
mobil.proc une fois lancé exécute le code de mob.proc qui fait suite à SUSPEND.
D'où l'affichage de "BBBBBBBBBB" et du saut de ligne .
Dans un programme réaliste un processus mobile se suspend , avant transfert , suite à la lecture d'un code particulier dans un canal .

14.4.1 Un exercice de synthèse

L'exemple suivant conclut ce chapitre en proposant au lecteur un petit programme qui illustre la mobilité des canaux et celle des processus . Ici un processus mobile est transféré à travers un canal mobile puis exécuté à sa réception.
Le processus generateur() crée le canal mobile et envoie ses deux extrémités aux processus client() et server() .
server() crée une instance mobil.print du processus mobile print() et l'envoie à client() à travers le canal mobile qui vient d'être crée .
client() réceptionne mobil.print dans mob.proc et lance celui ci en paralléle avec l'écriture de données dans le canal "canal" . mob.proc lit ces données et les affiche à l'écran .

```
-- ch14_5.occ
--
PROC TYPE M.PRT IS (CHAN BYTE in? , CHAN BYTE out!):

PROTOCOL PRT
  CASE
    want.prt
    done.prt
:

CHAN TYPE C.PRT
  MOBILE RECORD
    CHAN PRT  req?          :
    CHAN MOBILE M.PRT mproc! :
:


MOBILE PROC print(CHAN BYTE data.in? , CHAN BYTE scr.out!)IMPLEMENTS M.PRT
  BYTE data :
  SEQ i=0 FOR 10
    SEQ
      data.in ? data
      scr.out ! data
:

PROC generateur(CHAN C.PRT! canal1! , CHAN C.PRT? canal2!)
  C.PRT! cli  :
  C.PRT? svr  :
  SEQ
    cli, svr := MOBILE C.PRT
    PAR
      canal1! cli
      canal2! svr
:


PROC client(CHAN C.PRT! in.c?, CHAN BYTE  scr!)
  C.PRT! cli.end        :
  MOBILE M.PRT mob.proc  :
  INITIAL [10]BYTE data IS "AZERTYUIO*n":
  CHAN BYTE canal        :
```

```
  SEQ
    in.c? cli.end
    cli.end[req] ! want.prt
    cli.end[mproc]? mob.proc
    PAR
      SEQ i=0 FOR 10
        canal ! data[i]
      mob.proc(canal?, scr!)
:

PROC server(CHAN C.PRT? in.s?)
  C.PRT? svr.end            :
  MOBILE M.PRT mobil.print  :
  SEQ
    mobil.print := MOBILE print
    in.s ? svr.end
    svr.end[req]? CASE
      want.prt
        svr.end[mproc]! mobil.print
      done.prt
        SKIP
:

PROC main(CHAN BYTE scr!)
  CHAN C.PRT! request1  :
  CHAN C.PRT? request2  :
  PAR
    generateur(request1!, request2!)
    server(request2?)
    client(request1? , scr!)
:
```

Chapitre 15

Programmation système

Les processus Occam permettant la programmation système sont de simples "wrappers" sur leurs homologues POSIX SystemV d'UNIX.
Ci dessous nous donnons quelques exemples simples d'utilisation de ces modules.
Les lecteurs familiers de la programmation système en langage C sous UNIX se retrouveront tout de suite en terrain connu.

15.1 Les fichiers

Le programme ci dessous lit et affiche un fichier texte.
L'ouverture file.open() du fichier délivre un descripteur de fichier fd de type INT différent de -1 si elle réussit.
Ensuite ce descripteur est utilisé pour la lecture proprement dite file.read() dans un buffer qui est affiché puis le fichier est fermé par file.close().

```
-- ch15_1.occ
--
#INCLUDE "file.module"
#INCLUDE "course.module"


PROC main(CHAN BYTE kbd?, scr!)
  INT fd , size  , result   :
  [100]BYTE nom.fichier     :
  INITIAL [4096]BYTE buffer  IS [i=0 FOR 4096 | 0 ]       :
  SEQ
    --  La marque de fin de chaine suit la convention de'C'
    --  qui est 0
    --  On obtient un affichage propre du fichier texte
    --
    out.string("nom du fichier : ",0,scr!)
    flush(scr!)
    in.string(nom.fichier , size, 50 , kbd?, scr!)
    out.string("*ntaille lue : ",0,scr!)
    out.int(size, 0, scr!)
    flush(scr!)
```

```
    --
    file.open(nom.fichier , S.IRGRP , fd)
    IF
      fd < 0
        SEQ
          out.string("erreur open",0,scr!)
          STOP
      TRUE
        out.string("*nopen reussi*n",0,scr!)
    --
    file.read(fd, buffer, result)
    IF
      result < 0
        SEQ
          out.string("erreur read",0,scr!)
          STOP
      TRUE
        SEQ
          file.close(fd, result)
          out.string("read reussi*n",0,scr!)
          out.string(buffer,0,scr!)
          out.string("*nBY ...*n",0,scr)
:
```

15.2 La programmation réseaux

La programmation réseau s'appuie , comme sous UNIX, sur les sockets.
Les exemples , très simples , qui sont donnés montrent comment créer et initialiser un socket puis comment l'utiliser soit que l'on soit client ou que l'on soit serveur.

15.2.1 Les sockets

Un socket est une structure compactée dont les champs sont :

```
DATA TYPE SOCKET
  PACKED RECORD
    INT fd:             -- descripteur de fichier
    INT local.port:     -- numero de port local
    INT remote.port:    -- numero de port distant
    INT local.addr:     -- adresse IPv4 locale
    INT remote.addr:    -- adresse IPv4 distante
    INT s.family:       -- AF.INET
    INT s.type:         -- type de socket (SOCK.STREAM ou TCP, SOCK.DGRAM )
    INT s.proto:        -- protocole  IP  (IPPROTO.TCP  ou IPPROTO.UDP )
    INT error:          -- code d'erreur en cas d'echec de creation du socket
:
```

Le serveur

Dans cette première version, le serveur ne gère qu'une requè à la fois.
Il crée, en exécutant tcp.sock.creat(), un socket coté serveur qui accepte n'importe

quelle adresse IP (INADDR.ANY) sur le port (arbitraire) 3499.
Il se met en attente d'une requète en exécutant socket.listen().
Il entame alors une boucle indéfinie qui reconnait et gère les requètes des clients.
Dès qu'une requète est reconnue par socket.accept(), un socket client est crée sur lequel s'appuiera la connection. Il affiche l'adresse IP du client qui émet la requète puis envoie le message "Vous etes bien connectes sur le serveur XXX*n*n" au client en exécutant socket.write().
Le socket client est ensuite fermé par socket.close() et le serveur se met en attente d'une nouvelle requète.

On trouvera dans le paragraphe sur le forking au paragraphe 15.3.3 une seconde version plus réaliste de ce serveur.

```
-- ch15_2.Occ
--
#INCLUDE  "sock.module"
#USE "course.lib"


PROC tcp.sock.creat(SOCKET sock , INT result)
  VAL INT listen.addr IS INADDR.ANY :
  VAL INT listen.port IS 3499       :
  SEQ
    socket.create.listen.tcp(sock,listen.addr, listen.port, result)

:

PROC main(CHAN BYTE scr!)
  SOCKET sock.serv , sock.client :
  INT result            :
  VAL INT BACKLOG IS 10 :
  VAL []BYTE message IS "Vous etes bien connectes sur le serveur XXX*n*n":

  SEQ
    tcp.sock.creat(sock.serv, result)
    IF
      result < 0
        SEQ
          out.string("erreur creation socket *n",0,scr!)
          STOP
      TRUE
        SKIP
    --

    socket.listen(sock.serv, BACKLOG , result)
    IF
      result < 0
        SEQ
          out.string("erreur listen *n",0,scr!)
          STOP
      TRUE
        out.string("serveur en attente *n*n",0,scr!)

    --
```

```
  WHILE TRUE
    SEQ
      socket.accept(sock.serv,sock.client , result)
      IF
        result < 0
          SEQ
            out.string("erreur accept *n",0,scr!)
            STOP
        TRUE
          [20]BYTE ip.addr  :
          INT len           :
          SEQ
            out.string("accepte la connection de : ",0,scr!)
            socket.ip.of.addr(sock.client[remote.addr] , ip.addr , len, result)
            SEQ i=0 FOR len
              scr ! ip.addr[i]
            scr ! '*n'
      --
      SEQ
        socket.write(sock.client, message , result)
        IF
          result < 0
            SEQ
              out.string("erreur write *n",0,scr!)
              STOP
          TRUE
            SEQ
              socket.close(sock.client)
              out.string("ferme la connection ...*n",0,scr!)
:
```

Sur le terminal après acceptation d'une requète on peut voir (la requète est émise par un pc à l'adresse IP : 192.168.1.10) :

```
$ ./server
serveur en attente

accepte la connection de : 192.168.1.10
ferme la connection ...
```

Le client

Le client crée un socket sock.client par socket.creat(). Il initialise ensuite les champs sock.client[remote.port] (3499) donnant le port sur lequel le serveur est en attente et sock.client[remote.addr] (socket.addr.of.hoest()) qui fournit son adresse IP. Le client se connecte au serveur en exécutant socket.connect() puis lit (socket.read()) le message émis par ce dernier. Finalement le client se déconnecte en exécutant socket.close().

```
-- ch15_3.occ
--
#INCLUDE  "sock.module"
#USE "course.lib"

PROC affiche(VAL []BYTE msg , CHAN BYTE screen!)
  INT msg.size        :
  SEQ
    msg.size := SIZE msg
    SEQ i=0 FOR msg.size
      screen ! msg[i]
:


PROC main(CHAN BYTE scr!)
  SOCKET sock.client    :
  INT result            :
  [256]BYTE message     :
  SEQ
    socket.create(sock.client, AF.INET,SOCK.STREAM,IPPROTO.TCP)
    IF
      sock.client[fd] < 0
        SEQ
          out.string("erreur creation socket",0, scr!)
      TRUE
        SKIP
    out.string("creation socket OK *n",0, scr!)
    sock.client[remote.port]:= 3499
    -- le serveur AMD IP : 192.168.1.12
    socket.addr.of.host("192.168.1.12",sock.client[remote.addr],result)
    IF
      result < 0
        SEQ
          out.string("erreur adresse amd *n",0,scr!)
          STOP
      TRUE
        out.string("socket remote adress OK*n",0,scr!)
    --
    --  fin des initialisations
    --  connection au serveur puis lecture
    --
    socket.connect(sock.client, result)
    IF
      result < 0
        SEQ
          out.string("erreur connect hp *n",0,scr!)
```

```
          STOP
      TRUE
        out.string("socket connect OK *n",0,scr!)

    socket.read(sock.client,message, 256, result)
    IF
      result < 0
        SEQ
          out.string("erreur lecture  *n",0,scr!)
          STOP
      TRUE
        SEQ
          affiche(message, scr!)
          socket.close(sock.client)
          scr ! '*n'
:
```

Sur le terminal coté client on peut lire :

```
creation socket OK
socket remote adresse OK
socket connect OK
Vous etes bien connectes sur le serveur XXX
```

15.3 Le forking

Avec la directive FORK il est possible de lancer un processus en réponse à une requète asynchrone . On sait que la fonction fork() du language C est le moyen par lequel on crée un processus **lourd** sous UNIX. Un processus initial dit processus père lance fork() et crée un processus dit processus fils qui s'exécute en parallèle avec lui .Ces deux processus exécutent le même code et le processus fils se voit allouer une copie des ressources allouées au processus père. Le texte qui suit est typique d'un programme C qui utilise fork().

```
pid_t pid:

// pid_t est le type de donnees retourne par fork()
// apres appel a fork() pid = 0  cote fils et pid > 0 cote pere.
// ce qui permet de discriminer les codes du pere et du fils.

int main(){
  ...
  // le code du processus pere
  pid  = fork();
  switch (pid){
    case(-1){
       // erreur fork()
    }
    case(0){
      // le code du processus fils si pid vaut 0
    }
    default{
```

```
      // le code du processus pere si pid > 0
    }
  }
}
```

En occam-π la directive FORK permet à un processus 'pere' de lancer un processus 'fils' qui est un processus nommé. Si cette directive s'exécute dans un bloc FORKING le processus 'pere' est suspendu dans l'attente de la terminaison du processus 'fils' sinon les deux processus 'pere' et 'fils' s'exécutent en parallèle. Les processus nommés qui sont lançés par FORK ne peuvent avoir comme paramètres que des valeurs, des canaux (partagés ou mobiles), des barrieres mobiles pour se synchroniser.
Syntaxe :

```
forking             => FORK processus.nomme({0, fork.param})
                    => FORKING
                         {forked.process}

fork.param          => value.param
                    => shared.chan.param | mobile.chan.param
                    => mobile.barriere.param
```

NB Dans cette écriture la liste 'forked.process' contient au moins un processus lançé par FORK.

Dans l'exemple qui suit le processus main() entre dans une boucle de scrutation du clavier.
Si le caractère entré est 'q' il se termine sinon il lance par FORK le processus nommé printer() et lui passe la valeur (code ascii) entrée au clavier.
printer() affiche 5 fois cette valeur puis attend 1 seconde avant de s'achever. Si on frappe rapidement des caractères autres que 'q' on lance autant de processus dont les affichages s'entrelacent. Comme les processus sont lançés dans un bloc FORKING le processus main() ne reprend la main qu'après achèvement de ces derniers. On notera que le canal scr associé à l'écran est partagé entre main() et les processus printer() et donc l'accès à ce canal ne s'obtient que par un CLAIM .
Si on frappe "azertyq" au clavier on observe l'affichage de :

```
Starting
azaezraetzraetzraetzr et r t
BY ...
```

```
-- ch15_4.occ
--
#USE "course.lib"

PROC printer (VAL BYTE b, SHARED CHAN BYTE out!)
  VAL INT second IS 1000000:
  TIMER tim:
  INT t:
  SEQ
    SEQ i = 0 FOR 5
      SEQ
        CLAIM out!
          SEQ
            out ! b
            flush(out!)
        tim ? t
        tim ? AFTER t PLUS second
    CLAIM out!
      out! ' '
:

PROC main (CHAN BYTE kbd?, SHARED CHAN BYTE scr!)
  INITIAL BOOL running IS TRUE:
  SEQ
    CLAIM scr!
      out.string ("Starting*n", 0, scr!)
    FORKING
      WHILE running
        BYTE b:
        SEQ
          kbd ? b
          CASE b
            'q'
              running := FALSE
            ELSE
              FORK printer (b, scr!)
    CLAIM scr!
      out.string ("*nBY ...*n", 0, scr!)
:
```

Si on supprime le bloc FORKING alors le processus main() peut exécuter l'affichage de "*nBY ... *n" sans attendre la terminaison des processus printer().Si on entre "azertyq" au clavier alors un affichage possible est :

```
Starting
azertayzer
BY ...
tayzertayzertayzert y
```

15.3.1 Communication avec les canaux mobiles

Un canal partagé permet la communication entre un processus 'pere' et un processus 'fils' lançé par FORK. Dans l'exemple suivant la valeur 'a' est passée par main() au processus process() qui retourne la valeur ('a'+1) via le canal partagé s.canal

```
-- ch15_5.occ
--
PROC process(VAL BYTE b, SHARED CHAN BYTE out!)
  BYTE x:
  SEQ
    x := b + 1(BYTE)
    CLAIM out!
      out! x
:

PROC main(CHAN BYTE scr!)
  SHARED CHAN BYTE s.canal :
  BYTE x  :
  SEQ
    FORK process('a', s.canal!)
    CLAIM s.canal?
      s.canal? x
    scr ! x
    scr ! '*n'
:
```

Un canal mobile est le moyen qui permet , à deux processus lançés par FORK , de communiquer.
Premier exemple :

```
-- ch15_6.occ
--
PROTOCOL PRT
  CASE
    want.prt
    done.prt
:

CHAN TYPE M.RPRT
  MOBILE RECORD
    CHAN PRT  req?  :
    CHAN BYTE data! :
:


PROC client(M.RPRT! cli.end, SHARED CHAN BYTE visu!)
  BYTE car      :
  SEQ
    cli.end[req]!want.prt
    SEQ
      SEQ i=0 FOR 10
        SEQ
          cli.end[data]?car
          CLAIM visu!
            visu ! car
      CLAIM visu!
        visu ! '*n'
:
```

```
PROC server(M.RPRT? svr.end)
  VAL []BYTE data IS "HELLO ....":

  SEQ
    svr.end[req]?CASE
      want.prt
        SEQ i=0 FOR (SIZE data)
          svr.end[data]! data[i]
      done.prt
        SKIP
:


PROC main(SHARED CHAN BYTE scr!)
  M.RPRT! cli   :
  M.RPRT? svr   :
  SEQ
    cli, svr := MOBILE M.RPRT
    PAR
      -- les bouts du canal mobile sont communiques aux processus
      FORK server(svr)
      FORK client(cli, scr!)
:
```

Deuxieme exemple :
Cet exemple reprend le précédent mais les bouts du canal mobile sont tranférés à travers un canal qui **doit** être partagé.

```
-- ch15_7.occ
--
PROTOCOL PRT
  CASE
    want.prt
    done.prt
:

CHAN TYPE M.RPRT
  MOBILE RECORD
    CHAN PRT  req?  :
    CHAN BYTE data! :
:
```

```
PROC client(SHARED CHAN M.RPRT! canal1?, SHARED CHAN BYTE visu!)
  M.RPRT! cli.end :
  BYTE car      :
  SEQ
    CLAIM canal1?
      canal1? cli.end
    -- a recupere le bout client du canal mobile
    cli.end[req]!want.prt
    SEQ
      -- lit le message emis par le serveur sur
      -- le bout client du canal mobile et l'affiche a l'ecran
      SEQ i=0 FOR 10
        SEQ
          cli.end[data]?car
          CLAIM visu!
            visu ! car
      CLAIM visu!
        visu ! '*n'
:

PROC server(SHARED CHAN M.RPRT? canal2?)
  M.RPRT? svr.end:
  INITIAL [10]BYTE data IS "HELLO ....":

  SEQ
    CLAIM canal2?
      canal2? svr.end
    -- a  recupere le bout serv du canal mobile
    svr.end[req]?CASE
      -- ecrit []data sur le bout serveur du
      -- canal mobile
      want.prt
        SEQ i=0 FOR 10
          svr.end[data]! data[i]
      done.prt
        SKIP
:


PROC main(SHARED CHAN BYTE scr!)
  M.RPRT! cli   :
  M.RPRT? svr   :
  SHARED CHAN M.RPRT! requ.cli  :
  SHARED CHAN M.RPRT? requ.serv :
  SEQ
    -- creation des deux bouts d'un canal mobile
    cli, svr := MOBILE M.RPRT
    -- les bouts cli et serv du canal mobile
    -- sont envoyes a client() et server()
    --
    PAR
      CLAIM requ.cli!
        requ.cli! cli
      CLAIM requ.serv!
```

```
        requ.serv! svr
      --
      FORK server(requ.serv?)
      FORK client(requ.cli?, scr!)
:
```

15.3.2 Synchronisation avec les barrières mobiles

Les processus lançés par FORK() se synchronisent au moyen de barrières mobiles. Ils sont enrolés sur une barriere mobile dès lors que cette dernière est declarée en tant que paramètre. Les barrieres mobiles se comportent comme les barrières ordinaires mais sont astreintes aux contraintes liées aux ressources mobiles dans occam-π . On remarquera que l'écriture "INITIAL MOBILE BARRIER barriere IS MOBILE BARRIER :" déclare **et** initialise la barrière mobile "barriere".

Pour plus de précisions sur les barrières mobiles nous renvoyons à l'artickle de Fred Barnes et Peter Welch à :
`ftp://ftp.cs.ukc.ac.uk/pub/phw/sei-cmu/occam/papers/communicating-mobile-processes.pdf`

```
-- ch15_8.occ
--
#USE "course.lib"

PROC process(VAL INT index, SHARED CHAN BYTE ecran!, MOBILE BARRIER bar)
  TIMER t   :
  INT now   :
  INITIAL INT delai IS 300000:
  SEQ

    t? now
    t? AFTER now PLUS (index*delai)
    CLAIM ecran!
      SEQ
        flush( ecran!)
        ecran! 'a' + (BYTE index)
    SYNC bar                           -- synchronisation sur la barriere
    CLAIM ecran!
      SEQ
        flush(ecran!)
        ecran! 'A' + (BYTE index)
:

PROC main(SHARED CHAN BYTE scr!)
  SEQ
    FORKING
      MOBILE BARRIER barr:             -- declare la barriere mobile
      SEQ
        barr := MOBILE BARRIER         -- initialise la barriere mobile
        SEQ i=0 FOR 10
          FORK process(i, scr!, barr)  -- process est enrole sur barr
    CLAIM scr!
      out.string("*nBY ...*n",0, scr!)
:
```

Avant de se synchroniser les processus écrivent des minuscules à des intervalles de temps réguliers.
Après synchronisation, en exécutant SYNC, ils écrivent concuremment des majuscules qui donnent l'impression d'un affichage simultané.

Une autre manière, plus compacte, de déclarer et d'initialiser une barrière mobile dans l'exemple précédent.

```
PROC main(SHARED CHAN BYTE scr!)
  SEQ
    FORKING
      INITIAL MOBILE BARRIER barr IS MOBILE BARRIER :
      SEQ i=0 FOR 10
        FORK process(i, scr!, barr)
    CLAIM scr!
      out.string("*nBY ...*n",0, scr!)
:
```

15.3.3 Un serveur multi tâches

Ce serveur est la version multi tâches du serveur présenté au paragraphe 15.2.1.
La partie client est inchangée et est celle du paragraphe 15.2.1.
Pour chaque requète d'un client un socket spécifique est crée dont la gestion est confiée au processus request.respond() lançé par FORK. Ce processus, après écriture, est chargé de clore ce socket.
Comme on ne peut pas passer de paramètres par référence lors d'un FORK, ce processus, après avoir écrit dans le socket destiné au client, écrit dans le canal sock.release, partagé en ecriture, l'information qui permetra de fermer ce socket.

Le nombre maximum de connections autorisées est donné par l'entier max.connect.
Les sockets requises suite à une requète sont extraites du tableau []sock.client dimentionné à max.connect.
Le tableau de booléens []sock.ready également dimentionné à max.connect indique la disponibilité des sockets . Si sock.ready[i] vaut TRUE sock.client[i] est disponible , sinon il ne l'est pas.
La fonction find() a pour objet de retourner l'indice $i >= 0$ si sock.ready[i] vaut TRUE sinon elle retourne -1.
Dans sa boucle WHILE TRUE le serveur :
1 . Lit, sans être bloqué, le canal sock.release partagé en ècriture par tous les processus forkés. La lecture (éventuelle) donne l'indice sidx d'un socket sock.client[sidx] qui sera fermé et indiqué de nouveau comme disponible (sock.ready[sidex] := TRUE).
2 . Lance find() pour savoir si un socket d'indice sidx sera prêt pour faire face à une nouvelle requète. Dans l'état actuel du programme si sidx est < 0 le serveur exécute STOP. Si sidx est $>= 0$ socket.clien[sidx] sera choisi comme socket de connection à une requète d'un client et sock.ready[sidx] est mis FALSE.
3 . Une requète est acceptée et la transaction utilise socket.client[sidx].
4 . Le fork de request.respond() est exécuté avec comme paramètres :
L'indice sidx du socket client, le socket client sock.client[sidx], et le canal partagé en écriture sock.release .

Après cinq secondes le processus request.respond(), après avoir écrit dans sock.client[sidx], écrit sidx dans sock.release ce qui permet au processus “pére” (en 1.) de fermer ce socket.

```
-- ch15_9.occ
--
#INCLUDE  "sock.module"
#USE "course.lib"


VAL INT max.conect IS 10    :

PROC tcp.sock.creat(SOCKET sock , INT result)
  VAL INT listen.addr IS INADDR.ANY :
  VAL INT listen.port IS 3499       :
  SEQ
    socket.create.listen.tcp(sock,listen.addr, listen.port, result)

:

INT FUNCTION find(VAL []BOOL sr)
  BOOL trouve   :
  INT i, result :
  VALOF
    SEQ
      trouve := FALSE
      i := 0
      result := -1
      WHILE (NOT trouve ) AND (i < max.conect)
        IF
          sr[i]= TRUE
            SEQ
              result := i
              trouve := TRUE
          sr[i] = FALSE
            i := i+1
    RESULT result
:



PROC request.respond(VAL INT idx, VAL SOCKET sock, SHARED CHAN INT srl!)
  INITIAL [50]BYTE message IS "  :  Vous etes bien connectes sur le serveur AMD*n*n":
  INT result:
  SOCKET s  :
  --  le delai est de 5 secondes !!!
  VAL INT delai IS 5000000  :
  TIMER clock :
  INT now     :
  SEQ
    -- la signature de socket.write() exige que le 1er parametre
    -- soit une variable, d'ou la necessite de s.
    clock? now
    message[0] := '0' + (BYTE idx)
    s := sock
    clock? AFTER now PLUS  delai
    socket.write(s, message , result)
    --
```

```
    CLAIM srl!
      srl! idx
:

PROC main(CHAN BYTE scr!)
  SHARED! CHAN INT sock.release   :
  SOCKET sock.serv                :
  [max.conect]SOCKET sock.client :
  INITIAL [max.conect]BOOL sock.ready IS [i=0 FOR max.conect|  TRUE ] :
  INT sidx, result                :
  VAL INT BACKLOG IS 10 :
  SEQ
    tcp.sock.creat(sock.serv, result)
    IF
      result < 0
        SEQ
          out.string("erreur creation socket *n",0,scr!)
          STOP
      TRUE
        SKIP
    --

    socket.listen(sock.serv, BACKLOG , result)
    IF
      result < 0
        SEQ
          out.string("erreur listen *n",0,scr!)
          STOP
      TRUE
        out.string("serveur en attente *n*n",0,scr!)

    --  ************************************************

    WHILE TRUE
      SEQ
        PRI ALT
          sock.release? sidx
            SEQ
              socket.close(sock.client[sidx])
              sock.ready[sidx] := TRUE
          TRUE & SKIP
            SKIP
        --
        sidx := find(sock.ready)
        IF
          sidx < 0
            SEQ
              out.string("max connections depasse *n",0, scr!)
              STOP
          sidx >= 0
            sock.ready[sidx] := FALSE

        socket.accept(sock.serv,sock.client[sidx] , result)
        IF
```

```
        result < 0
          SEQ
            out.string("erreur accept *n",0,scr!)
            STOP
        TRUE
          [20]BYTE ip.addr   :
          INT len            :
          SEQ
            out.string("accepte la connection de : ",0,scr!)
            socket.ip.of.addr(sock.client[sidx][remote.addr], ip.addr, len, result)
            SEQ i=0 FOR len
              scr ! ip.addr[i]
            scr ! '*n'
            out.string("connection numero :",0, scr!)
            out.int(sidx,0, scr!)
            scr! '*n'
            --
            FORK request.respond(sidx, sock.client[sidx], sock.release!)

:
```

15.4 CIF et l'interface avec le language C

CIF (C Interface) est l'interface entre occam-π et C proposé par Kroc.
La nécessité d'une telle interface nait des besoins d'avoir des portions de code plus efficace pour des applications sensibles mais aussi, notamment en robotique, de pouvoir incorporer des sources C liées à la gestion de capteurs.

Bien que la documentation à ce jour (premier semestre 2011) sur CIF ne soit pas actualisée les nombreux exemples fournis avec la dernière distribution 1.5 de Kroc et rassemblés dans le répertoire : ../kroc-svn/modules/cif si l'installation été faite par svn. (cf l'annexe consacrée à kroc) sont très pertinents et permettent à celui qui a quelque expérience de la programmation système en C de s'y retrouver.

Documentation

L'article de Fred Barnes :
Interfacing C and occam-pi (Computing Process Architectures 2005) téléchargeable à :
`http://www.cs.kent.ac.uk/pubs/2005/2271/index.html`

L'API de CIF "The occam-pi C interface" de 2005, bien qu'obsolète, reste utile. Elle est téléchargeable à :
`http://www.cs.kent.ac.uk/projects/ofa/kroc/cif.html`

Nous appelerons processus CIF toute fonction C gérée par CIF et vue comme un processus par occam-π .
Les processus CIF sont autonomes et ne communiquent avec les autres processus (CIF ou pas) que par le biais des canaux.

15.4.1 Un premier exemple

Ce premier exemple implémante un appel à un processus écrit en C et qui calcule (trivialement) $n!$. Il est constitué de deux fichiers sources distincts. :
fact.occ qui est le fichier source en occam-π .
fact.c qui est le fichier source en C.
Ces deux fichiers, supposés dans le même répertoire, sont compilés par kroc en exécutant :
> kroc fact.occ fact.c.
Un exécutable fact est alors généré dans le même répertoire.

Le fichier Occam :

Tout programme Occam utilisant CIF doit mentionner #INCLUDE "cif.module".
Tout processus CIF doit faire l'objet d'une déclaration #PRAGMA dans l'en-tête du programme occam-π .
Il y a autant de #PRAGMA que de processus CIF déclarés.
Par exemple dans l'exemple proposé :

```
#PRAGMA EXTERNAL "PROC CIF.my.factoriel (CHAN INT in?, out!) = 1024"
```

"CIF.my.factoriel" fait référence à la fonction C "my_factoriel" (le '.' n'est pas autorisé dans un identificateur C mais le '_' l'est . Kroc fait la conversion).
CHAN INT in ?, out ! fait référence aux canaux qui établissent la communication entre le processus occam-π main() et le processus CIF my_factoriel() qui utilise in ? en lecture et out ! en écriture.
Attention !
L'écriture du PRAGMA pourrait faire penser que les canaux in ? et out ! sont passés comme des paramètres C ordinaires.
Il n'en est **rien**. Comme on le verra ci après le prototype en C de **tout** processus CIF est "void processus_cif (Workspace)" ce qui exclu que in ?, et out ! soient passés comme des paramètres C ordinaires sur une pile.
La valeur 1024 qui est affectée a CIF.my.factoriel dans le PRAGMA est en relation avec l'espace de travail attribué au processus CIF. L'article de Fred Barnes cité plus haut donne des précisions sur sa définition.

Le processus main() est d' une écriture standart. En particulier le processus CIF est traité comme tout processus occam-π . Un entier n est lu au clavier, il est écrit dans le canal c.in, puis est lu par CIF.my.factoriel() dans ce même canal . CIF.my.factoriel() calcule $n!$ puis l'écrit dans c.out. La lecture de c.out par le main() permet l'affichage du résultat.

```
-- ch15_10.occ
--
--
--fact.occ

#INCLUDE "cif.module"
#INCLUDE "course.module"

#PRAGMA EXTERNAL "PROC CIF.my.factoriel (CHAN INT in?, out!) = 1024"
```

```
PROC main (CHAN BYTE kybd?, scr!)
  CHAN INT c.in, c.out      :
  INT n  ,f                 :
  SEQ
    out.string("calcul de factorielle n *n",0 ,scr!)
    out.string("entrer n : ",0 , scr!)
    in.int(n, 5, kybd?, scr!)
    PAR
      c.in! n
      CIF.my.factoriel (c.in?, c.out!)
      c.out? f
    out.string("*nfactorielle  n : ",0 , scr!)
    out.int(f, 5, scr!)
    scr! '*n'
:
```

Le fichier C :

Il importe de conserver toutes les directives #include et en particulier #include <cif.h>.
Toute fonction C gérée par CIF est de prototype “void identificateur(Workspace)” où Workspace est un pointeur sur un entier défini dans ccsp_cif.h.
A l'exécution CIF passe en paramètre à my_factoriel() wptr qui pointe sur l'espace de travail qui lui est alloué. L'accès aux structures de données définissant les canaux in et out se fait alors en donnant leurs offsets relatifs à la valeur de wptr. .(0 pour le premier canal, 1 pour le second).
Les fonctions ProcGetParam() permettent alors l'accès aux canaux fact_in et fact_out de type (Channel *).
La lecture (resp l'ecriture) dans ces canaux se fait par ChanInInt() (resp ChanOutInt()).

```
//
//  fact.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <cif.h>

void my_factoriel (Workspace wptr){
  Channel *fact_in   = ProcGetParam (wptr, 0, Channel *);
  Channel *fact_out  = ProcGetParam (wptr, 1, Channel *);
  int   n , j, fact                     ;

  ChanInInt(wptr, fact_in, &n)          ;

  fact = 1                              ;
  for(j=1; j <= n ; j++) fact = fact * j  ;

  ChanOutInt(wptr, fact_out, fact)            ;
}
```

15.4.2 Autre version du premier exemple

Dans cette variante de l'exemple précédent les entrées sorties sont entièrement gérées par le processus CIF grace à des wrappers fournis par la fonction ExternalCallN () qui, dans l'exemple qui suit, permet de récupérer fprintf() et fscanf() en s'affranchissant des canaux occam liés au clavier et à l'écran. On remarque que dans le #PRAGMA du texte Occam il n'est fait référence à aucun paramètre.

```
-- ch15_11.occ
--  factb.occ
--

#INCLUDE "cif.module"
#PRAGMA EXTERNAL "PROC CIF.my.factoriel () = 1024"

PROC main ()
  CIF.my.factoriel()
:
```

En fait, comme le montre déja cet exemple très simple un programme occam-π peut très bien n'être constitué que de processus CIF.

```
// ch15_11.c
//

//  factb.c
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <cif.h>

void my_factoriel (Workspace wptr){
  int   n , j, fact                                  ;

  ExternalCallN (fprintf, 3, stdout, "entrer n : ") ;
  ExternalCallN (fscanf , 3, stdin ,"%d", &n)       ;

  fact = 1                                           ;
  for(j=1; j <= n ; j++) fact = fact * j             ;

  ExternalCallN (fprintf, 3, stdout, "la valeur de n! est  : %d\n", fact);
}
```

15.4.3 Parallélisme (1)

Dans cet exemple deux processus CIF proc1 et proc2 s'exécutent en parallèle et communiquent dans le canal canal1.

proc1 écrit 'A' dans le canal1.
proc2 lit ('A') dans canal1 ajoute 1 à la valeur lue et écrit le résultat ('B') dans le canal scr lié à l'écran.
On remarque l'existence de deux #PRAGMA correspondant aux déclarations de deux processus CIF.
On notera également les deux fonctions ChanInChar() et ChanOutChar() qui permettent la lecture (resp l'écriture) d'un BYTE par un processus CIF à travers un canal déclaré dans le programme occam-π .

Le programme Occam :

```
-- ch15_12.occ
--
#INCLUDE "cif.module"

#PRAGMA EXTERNAL "PROC CIF.proc1 (CHAN BYTE out!) = 1024"
#PRAGMA EXTERNAL "PROC CIF.proc2 (CHAN BYTE in?, out!) = 1024"

PROC main (CHAN BYTE scr!)
  CHAN BYTE canal1 :
  SEQ
    PAR
      CIF.proc1 (canal1!)
      CIF.proc2 (canal1?, scr!)
    scr! '*n'
:
```

Le programme C :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <cif.h>

void proc1 (Workspace wptr){
  Channel *out_p1 = ProcGetParam (wptr, 0, Channel *);
  char c = 'A'  ;

  ChanOutChar(wptr, out_p1, c)    ;
}


void proc2 (Workspace wptr){
  Channel *in_p2  = ProcGetParam (wptr, 0, Channel *);
  Channel *out_p2 = ProcGetParam (wptr, 1, Channel *);
  char c   ;

  ChanInChar (wptr, in_p2 , &c)      ;
  c = c+1                            ;
  ChanOutChar(wptr, out_p2, c)       ;
}
```

15.4.4 Parallélisme (2)

On reprend le programme qui vient d'être exposé. En particulier proc1() et proc2() restent inchangés. Par contre ces derniers ne sont plus vus par le programme occam-π . Ils sont lançés en parallèle par le processus CIF main_proc() qui seul est vu par occam-π qui lui passe en paramètre le canal scr ! lié a l'écran. On notera que l'espace de travail alloué à main_proc() est de 4096 ce qui sera justifié par la suite.

Le programme Occam :

```
-- ch15_13.occ
--
#INCLUDE "cif.module"
#PRAGMA EXTERNAL "PROC CIF.main.proc (CHAN BYTE out!) = 4096"

PROC main (CHAN BYTE scr!)
  SEQ
    CIF.main.proc (scr!)
    scr! '*n'
:
```

Le programme C :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <cif.h>

void proc1 (Workspace wptr){
  Channel *out_p1 = ProcGetParam (wptr, 0, Channel *);
  char c = 'A'  ;

  ChanOutChar(wptr, out_p1, c)    ;
}

void proc2 (Workspace wptr){
  Channel *in_p2  = ProcGetParam (wptr, 0, Channel *);
  Channel *out_p2 = ProcGetParam (wptr, 1, Channel *);
  char c          ;

  ChanInChar (wptr, in_p2 , &c)     ;
  c = c+1                           ;
  ChanOutChar(wptr, out_p2, c)      ;
}

void main_proc(Workspace wptr){
  Channel *out_scr = ProcGetParam(wptr, 0, Channel *) ;
  Channel canal2            ;
  Workspace wptr1, wptr2      ;
  word stack1[WORKSPACE_SIZE (1, 1024)]  ;
  word stack2[WORKSPACE_SIZE (2, 1024)]  ;
```

```
  //
  ChanInit(wptr, &canal2)     ;

  wptr1 = LightProcInit(wptr, stack1, 1, 1024)  ;
  ProcParam(wptr, wptr1, 0, &canal2)            ;
  //
  wptr2 = LightProcInit(wptr, stack2, 2, 1024)  ;
  ProcParam(wptr, wptr2, 0, &canal2)            ;
  ProcParam(wptr, wptr2, 1, out_scr)            ;
  //
  ProcPar(wptr, 2, wptr1, proc1, wptr2, proc2)  ;
}
```

On voit que main_proc() :
Récupère dans **son** espace de travail pointé par wptr l'accès au canal scr dans out_scr.
Déclare le canal canal2 qui joue le même rôle que canal1 de l'exemple précédent .
Déclare les deux pointeurs wptr1 et wptr2 qui seront associés aux deux futurs processus CIF de codes proc1() et proc2().
Déclare deux tableaux stack1 et stack2 de dimension 1024 (bytes) qui seront les piles des processus CIF de codes proc1() et proc2().
Initialise , dans son espace de travail, le canal canal2. On remarque que canal2 n'est associé à aucun protocole.
Initialise wptr1 et wptr2 . La pile et les paramètres de canaux sont alors assignés à ces derniers.
On notera que ProcParam(), qui joue le même rôle que ProcGetParam(), transmet les paramètres liés aux canaux à partir du contexte de main_proc() pointé par wptr vers les contextes pointés par wptr1 (resp wptr2) qui seront associés à proc1() (resp proc2()).
La liaison finale entre les contextes wptr1, wptr2 et les codes proc1(), proc2() se fait au moment de leur lancement en parallèle par ProcPar() à partir du contexte wptr de main_proc.
On comprend mieux qu'une taille de 4096 bytes soit allouée au contexte de main_proc dans le PRAGMA du texte occam-π .
En effet chacun des processus CIF proc1() (resp proc2()) s'est vu alloué un espace de travail de 1024 par LightProcInit() doublé d'une de pile de taille également de 1024.

15.4.5 Les canaux mobiles

Pour illustrer la façon dont les canaux mobiles sont gérés par CIF nous reprenons l'exemple 13.1.3 du chapitre 13.
Le programme en Occam est identique en tous points à celui de l'exemple cité à ceci près que le processus client() devient maintenant CIF.client(). Il y a également la présence de la déclaration PRAGMA qui est nouvelle.

```
-- ch15_14.occ
--
#INCLUDE "cif.module"

CHAN TYPE M.CLRV
  MOBILE RECORD
    CHAN BYTE req?  :
    CHAN BYTE resp! :
```

```
:

#PRAGMA EXTERNAL "PROC CIF.client(M.CLRV! resp, CHAN BYTE out!)=1024"

PROC serveur(M.CLRV? canal)
  BYTE requete  :
  SEQ
    canal[req]? requete       -- attend une requete
    canal[resp]!requete + 1   -- retourne la reponse
:

PROC main(CHAN BYTE scr!)
  M.CLRV? serv  :
  M.CLRV! cli   :
  SEQ
    serv, cli := MOBILE M.CLRV
    PAR
      CIF.client(cli , scr!)
      serveur(serv)
:
```

Le programme C :

La structure du programme en C est la même qu'en occam-π . Comme dans les exemples précédents il convient, en premier lieu, de récuperer les structures C qui acceuillent les canaux qui sont passées “en paramètres” par le biais du pointeur wptr sur la structure allouée au processus CIF lors de son lancement.
Le fait nouveau est que pour les canaux mobiles cette structure est de type **mt_cb_t** au lieu de Channel pour les canaux ordinaires. Comme un canal mobile regroupe en général (ici deux) plusieurs canaux ces derniers sont repérés par leurs offsets donnés par M_CRLV_req , M_CRLV_resp. (les identificateurs donnés à ces offsets n'ont aucune importance (ils valent 0 et 1) mais rapellent, par soucis de lisibilité, ceux des champs du MOBILE RECORD d'occam-π .

```
// ch15_15.c
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <unistd.h>

#include <cif.h>

enum M_CRLV_channels {
  M_CRLV_req ,
  M_CRLV_resp
};

void client (Workspace wptr){
  mt_cb_t *link_cli = ProcGetParam(wptr, 0, mt_cb_t *);
```

```
  Channel *out_scr  = ProcGetParam(wptr, 1, Channel *);
  char reponse        ;

    ChanOutChar (wptr, &(link_cli->channels[M_CRLV_req]), 'X')      ; // emet une requete
    ChanInChar (wptr,  &(link_cli->channels[M_CRLV_resp]), &reponse); // attend la reponse
    //
    ChanOutChar(wptr, out_scr, reponse) ;                             // affiche la reponse
    ChanOutChar(wptr, out_scr, '\n')    ;
}
```

Chapitre 16

Programmation graphique

16.1 Introduction

Tout programme graphique doit exécuter en parallèle le processus raster.display() et un processus défini par le programmeur (ici draw()) qui communiquent par les canaux raster.in et raster.out.
raster.display() est le processus de base fourni par le module “sdlraster.module” qui communique directement avec le hardware.
Il affiche une fenêtre dont l'intitulé est fourni en premier paramètre (“droites”) suivi de sa largeur (dim), de sa hauteur(dim), du nombre d'affichage à gérer (ici 1) suivi des deux canaux de communication associés aux flux avec la mémoire d'écran utilisateur qui est controlée par draw().
Dans l'exemple qui est donné raster.display.simple() qui ne gère pas d'événements liés au clavier ou à la souris est utilisé à la place de raster.display().

```
  PAR
    raster.display.simple ("droites",dim ,dim , 1, raster.out?, raster.in!)
    draw (raster.in?, raster.out!)
:
```

La mémoire d'écran utilisateur est une variable de type RASTER, locale à draw(), en relation avec le frame buffer par le biais des canaux raster.in et raster.out .
Nous allons montrer dans un premier exemple que cette mémoire utilisateur peut être directement programmée. C'est en réalité un tableau MOBILE de type INT à deux dimensions, chaque élément de tableau correspondant à un pixel affecté d'une couleur.
L'affichage horizontal constitue l'axe des X positifs et l'affichage vertical celui des Y positifs. Le coin supérieur gauche de la fenêtre a pour coordonnées $(0, 0)$.
Si [][]rast est la mémoire d'écran utilisateur le point de coordonnées (x,y) est affecté d'une couleur (bleue par exemple) par rast[y][x] := COLOUR.BLUE.

Premier exemple :

Dans cet exemple des droites horizontales de différentes couleurs sont tracées en écrivant directement dans la mémoire d'ecran rast.

```
-- ch16_1.occ
--
#INCLUDE "sdlraster.module"
#INCLUDE "rastergraphics.module"


VAL INT dim IS 520      :

VAL [6]INT couleur IS [COLOUR.RED, COLOUR.MAGENTA, COLOUR.SKY, COLOUR.ORANGE,
                       COLOUR.YELLOW, COLOUR.GREEN]:


PROC clear (RASTER r)
  SEQ y = 0 FOR dim
    SEQ x = 0 FOR dim
      r[y][x] := COLOUR.BLACK
: --  clear()


PROC draw (CHAN RASTER in?, out!)
  RASTER rast               :
  INITIAL INT i  IS 0       :
  SEQ
    in? rast
    clear(rast)
    SEQ y=0 FOR 26 STEP 20  -- les droites sont espacees de 20 pixels
      SEQ
        i := (i+1) REM 6
        SEQ x=0 FOR 520
          SEQ
            rast[y][x]    := couleur[i]
            rast[y+1][x]  := couleur[i]
    out! rast
: --  draw()


PROC main ()
  CHAN RASTER raster.in, raster.out:
  PAR
    raster.display.simple ("droites", dim, dim, 1, raster.out?, raster.in!)
    draw (raster.in?, raster.out!)
:
```

Deuxième exemple :

Le même programme mais en utilisant la fonction graphique draw.horizontal.line().

```
-- ch16_2.occ
--
#INCLUDE "sdlraster.module"
#INCLUDE "rastergraphics.module"
```

```
VAL INT dim IS 520      :

VAL [6]INT couleur IS [COLOUR.RED, COLOUR.MAGENTA, COLOUR.SKY, COLOUR.ORANGE,
                        COLOUR.YELLOW, COLOUR.GREEN]:


PROC clear (RASTER r)
  SEQ y = 0 FOR dim
    SEQ x = 0 FOR dim
      r[y][x] := COLOUR.BLACK
:


PROC draw (CHAN RASTER in?, out!)
  RASTER rast               :
  INITIAL INT i  IS 0       :
  SEQ
    in? rast
    clear(rast)
    SEQ y=0 FOR 26 STEP 20
      SEQ
        i := (i+1) REM 6
        SEQ x=0 FOR 520
          draw.horizontal.line (x, y,dim, couleur[i], rast)
    out! rast
: --  draw()


PROC main ()
  CHAN RASTER raster.in, raster.out:
  PAR
    raster.display.simple ("droites" ,dim ,dim , 1, raster.out?, raster.in!)
    draw (raster.in?, raster.out!)
:
```

16.2 La courbe de Hilbert

Le programme hilbert.occ permet de tracer une courbe de Hilbert jusqu'à l'ordre 9 autorisé par la résolution de l'écran.
Elle utilise les tracés de segments de droites horizontaux ou verticaux fournis par le module "rastergraphics.module".

Les paramétres de draw() sont , à partir du deuxiéme, l'ordre de la courbe , les coordonnées du coin superieur gauche de la courbe, la longueur d'un traçé rectiligne (horizontal ou vertical) lié à la courbe ,la couleur de la courbe , les canaux de communication avec le frame buffer .
Le premier paramétre "type" de hilb() est lié au type de courbe qui sera écrit en premier. En général le traçé de la courbe de Hilbert utilise une récursion mutuelle entre quatre types de courbes (des ⊔ ne diffèrant que par leur orientation). Occam ne permettant pas la récursion mutuelle cette dernière a été transformée en récursion simple par l'artifice de ce paramètre "type".

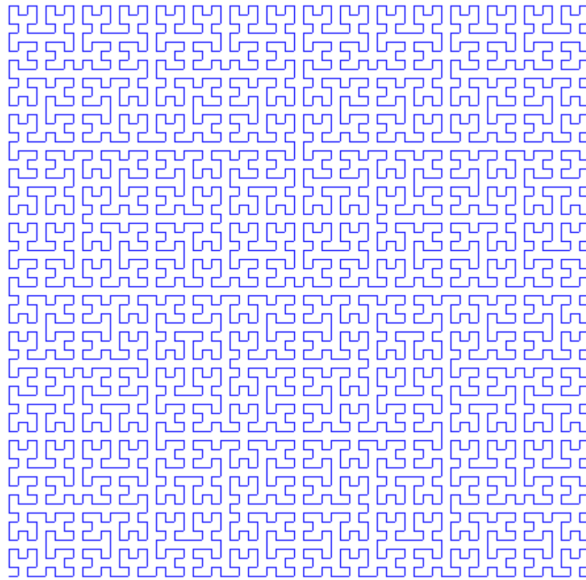


FIGURE 16.1 – Courbe de Hilbert d'ordre 7

Le programme :

```
-- ch16_3.occ
--
--  hilbert.occ

#INCLUDE "sdlraster.module"
#INCLUDE "rastergraphics.module"

#USE "course.lib"

-- shl (i) calcule 2**i

INT FUNCTION shl(VAL INT i) IS (1 << i):


PROC print.str(VAL BYTE x,y, VAL []BYTE string, CHAN BYTE out!)
  SEQ
    cursor.x.y(x,y,out!)
    out.string(string,0, out!)
    flush(out!)
:


REC PROC hilb (VAL INT type, n ,x , y ,t,color, RASTER rst)
  INT d :
  SEQ
    d := t*shl(n-1)
    CASE type
```

```
1
  IF
    n > 1
      SEQ
        hilb(1,n-1,x,y,t,color, rst)
        draw.vertical.line(x,y+(d-t),t, color, rst)
        hilb(2,n-1,x,y+d,t,color, rst)
        draw.horizontal.line(x+(d-t),y+(d-t),t, color, rst)
        hilb(1,n-1,x+d,y,t,color, rst)
        draw.vertical.line(x+((2*d)-t),y+(d-t),t, color, rst )
        hilb(3,n-1,x+d,y+d,t,color, rst)
    n = 1
      SEQ
        draw.vertical.line(x   ,y  ,t,color, rst)
        draw.horizontal.line(x ,y , t,color, rst)
        draw.vertical.line(x+t ,y , t,color, rst)
2
  IF
    n > 1
      SEQ
        hilb(4,n-1,x,y,t,color, rst)
        draw.horizontal.line(x+(d-t),y,t, color, rst)
        hilb(1,n-1,x,y+d,t,color, rst)
        draw.vertical.line(x+d,y+(d-t),t, color, rst )
        hilb(2,n-1,x+d,y,t,color, rst)
        draw.horizontal.line(x+(d-t),y+((2*d)-t),t, color, rst)
        hilb(2,n-1,x+d,y+d,t,color, rst)

    n = 1
      SEQ
        draw.horizontal.line(x  ,y   , t,color, rst )
        draw.vertical.line(x+t  ,y   , t,color, rst )
        draw.horizontal.line(x  ,y+t , t,color, rst )

3
  IF
    n > 1
      SEQ
        hilb(3,n-1,x,y,t,color, rst)
        draw.horizontal.line(x+(d-t),y,t, color, rst)
        hilb(3,n-1,x,y+d,t,color, rst)
        draw.vertical.line(x+(d-t),y+(d-t),t, color, rst )
        hilb(4,n-1,x+d,y,t,color, rst)
        draw.horizontal.line(x+(d-t),y+((2*d)-t),t, color, rst)
        hilb(1,n-1,x+d,y+d,t,color, rst)
    n = 1
      SEQ
        draw.horizontal.line(x    ,y   , t,color, rst)
        draw.vertical.line(x      ,y   , t,color, rst)
        draw.horizontal.line(x    ,y+t , t,color, rst)

4
  IF
    n > 1
```

```
            SEQ
              hilb(2,n-1,x,y,t,color, rst)
              draw.vertical.line(x,y+(d-t),t, color, rst )
              hilb(4,n-1,x,y+d,t,color, rst)
              draw.horizontal.line(x+(d-t),y+d,t, color, rst)
              hilb(3,n-1,x+d,y,t,color, rst)
              draw.vertical.line(x+((2*d)-t),y+(d-t),t, color, rst )
              hilb(4,n-1,x+d,y+d,t,color, rst)
          n = 1
            SEQ
              draw.vertical.line(x,     y   ,t,  color, rst)
              draw.horizontal.line(x   , y+t , t, color, rst)
              draw.vertical.line( x+t , y   , t, color, rst)
:


PROC draw (VAL INT ordre,x,y,trait,color, CHAN RASTER in?, out!)
  RASTER rst  :
  WHILE TRUE
    SEQ
      in? rst
      raster.clear(rst , COLOUR.WHITE)
      hilb(1, ordre ,x,y,trait,color, rst)
      out! rst
:

PROC main(CHAN BYTE kbd?, scr!)
  VAL INT color IS COLOUR.BLUE      :
  VAL INT WIDTH IS 800              :
  CHAN RASTER raster.in, raster.out :
  INT line, ordre                   :
  BOOL trace                        :

  SEQ
    erase.screen(scr!)
    print.str(2,2,"Courbe de Hilbert", scr!)
    print.str(2,4,"Ordre la courbe (1..9) : ", scr!)
    in.int(ordre, 4, kbd?, scr!)
    trace := TRUE
    IF
      ordre < 10
        -- line donne la longueur d'un segment de droite associe a l'ordre
        SEQ
          line := shl(9- ordre)
          PAR
            raster.display.simple ("Courbe de Hilbert", WIDTH, WIDTH, 1, raster.out?, raster.in
            draw (ordre, 100,100,line,color, raster.in?, raster.out!)
      TRUE
        SEQ
          print.str(2,6,"Impossible de tracer la courbe", scr!)
          print.str(2,7,"Entrer un ordre compris entre 1 et 9", scr!)
          scr! '*n'
:
```

16.3 Le problème des deux corps

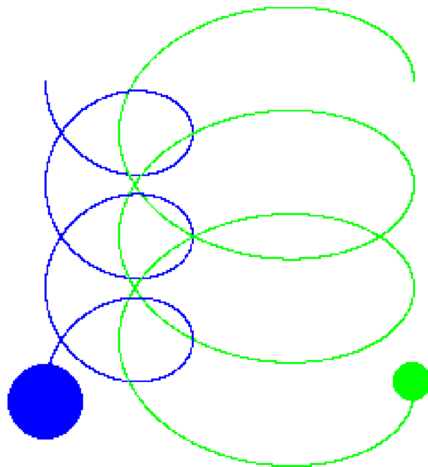


FIGURE 16.2 – Deux corps

Dans cet exemple deux processus, deux corps célestes, échangent en permanence leur cordonnées par le biais de canaux, en déduisent la distance qui les séparent et, en accord avec la loi de Newton sur les forces auquelles ils sont soumis, modifient leur accélération,leur vitesse et finalement leur position.
En fonction des positions , des masses et des vitesses initiales des deux corps toutes les trajectoires, parfois complexes (voir la figure 16.2 ci dessus) peuvent être obtenues.
La loi de Newton :
$\vec{F_{01}}$ est la force exercée par le corps de masse M_0 sur le corps de masse M_1.
$\vec{u_{01}}$ est le vecteur unitaire d'origine le corps de masse M_0 et dirigé vers le corps de masse M_1 et porté par la droite joignant les centres des deux corps distants de d.
On sait que le corps de masse M_1 exerce sur le corps de masse M_0 une force $\vec{F_{10}}$ égale et opposée à $\vec{F_{01}}$.

$$\vec{F_{01}} = k * \frac{M_0 * M_1}{d^2} \vec{u_{01}}$$

Quelques commentaires .
L'interface utilisateur utilise le module "convert.lib" pour pouvoir lire des flottants sur 64 bits . (cf le processus read.real64())
Chacun des deux corps lit et communique sa position à l'autre en utilisant les canaux body.b10 et body.b01.
Ensuite , par un appel à gravity() il détermine son accéleration puis sa vitesse et finalement ses nouvelles coordonnées .
Cela fait il transmet ces derniéres à draw par l'un des canaux coord0 (resp coord1).
draw() rafraichit l'écran puis la mémoire d'écran utilisateur raster[][] considérée comme un tableau à deux dimensions est initialisée de la trajectoire des deux corps sauvegardée suite à l'appel précédent dans le tableau trajectoire[][].

draw() lit ensuite les coordonnées des deux corps , et procéde à la determination de leur traçé (fill.circle()) de même que celui de leur nouvelle position dans trajectoire[][]. Finalement la mémoire d'écran utilisateur raster est écrite dans le frame buffer pour affichage à travers le canal raster.out.

Le programme :

Pour des raisons de modularité mais aussi parcequ'une part importante du code est réutilisée dans l'exemple suivant (mise sur orbite d'un satellite) le programme est réparti sur deux fichiers textes : gravite.occ et deux.occ.

Le fichier gravite.occ :

Le fichier gravite.occ regroupe les structures de données, deux fonctions utilitaires, la fonction gravity() et le processus body().
gravity() permet à chaque corps de calculer son accélération en fonction de sa masse et de la distance qui le sépare de l'autre corps.
Le processus body() transmet ses coordonnées à l'autre corps et recueille les siennes. Il fait appel à gravity() pour actualiser sa position et l'envoie au processus draw() pour affichage.

gravite.occ :

```
-- ch16_4.occ
--
--  gravite.occ


#INCLUDE "course.module"
#INCLUDE "convert.module"



DATA TYPE CORDINATE
  RECORD
    REAL64 x  :
    REAL64 y  :
:

DATA TYPE SPEED
  RECORD
    REAL64 vx :
    REAL64 vy :
:

DATA TYPE ACCELER
  RECORD
    REAL64 gx :
    REAL64 gy :
:


VAL INT WIDTH IS 800         :
```

```
PROC print.str(VAL BYTE x,y, VAL []BYTE string, CHAN BYTE out!)
  SEQ
    cursor.x.y(x,y,out!)
    out.string(string,0, out!)
    flush(out!)
:

PROC read.real64(REAL64 x , CHAN BYTE in?, out!)
  BOOL erreur     :
  [20]BYTE string :
  INT len         :
  REAL64 u        :
  SEQ
    in.string(string,len,20,in?, out!)
    STRINGTOREAL64(erreur, u, string)
    flush(out!)
    x := u
:

ACCELER FUNCTION gravity(VAL CORDINATE c1,c2, VAL REAL64 m)
  VAL REAL64 gravite IS 0.008  :
  CORDINATE unitaire           :
  REAL64 sdelta, delta2        :
  ACCELER grave                :
  VALOF
    SEQ
      unitaire[x] := c2[x] - c1[x]
      unitaire[y] := c2[y] - c1[y]
      delta2 := (unitaire[x]*unitaire[x]) + (unitaire[y]*unitaire[y])
      sdelta := DSQRT( delta2)
      unitaire[x] := unitaire[x] / sdelta
      unitaire[y] := unitaire[y] / sdelta
      grave[gx] :=  ((gravite * unitaire[x])*m) / delta2
      grave[gy] :=  ((gravite * unitaire[y])*m) / delta2
    RESULT grave
:


PROC body(CHAN CORDINATE c.in?, c.out!, VAL CORDINATE c, VAL SPEED v,
  VAL REAL64 mass, VAL REAL64 dt,CHAN CORDINATE out.raster!, BARRIER sync)
  INITIAL CORDINATE  c1 IS c    :
  INITIAL SPEED      v1 IS v    :
  INITIAL ACCELER    g1 IS [0.0, 0.0] :
  CORDINATE c0                  :
  WHILE TRUE
    SEQ
      PAR
        c.in?  c0
        c.out! c1

      --  nouvelle acceleration
      g1 := gravity(c1,c0,mass)

      -- nouvelle vitesse
```

```
      v1[vx] := v1[vx] + (g1[gx]*dt)
      v1[vy] := v1[vy] + (g1[gy]*dt)

      -- nouvelles coordonnees
      c1[x] := c1[x] + (v1[vx]*dt)
      c1[y] := c1[y] + (v1[vy]*dt)
      --
      out.raster! c1
      SYNC sync
:
```

Le fichier deux.occ :
Il regroupe le processus main() qui lance en parallèle trois processus : les deux corps après que leurs positions initiales et leurs vitesses initiales aient été lues au clavier ainsi que le processus de traçé des trajectoires draw().

deux.occ :

```
-- ch16_5.occ
--
--  deux.occ

#INCLUDE "gravite.occ"

#INCLUDE "sdlraster.module"
#INCLUDE "rastergraphics.module"




PROC draw (CHAN CORDINATE chan0?, chan1?,VAL REAL64 m0,m1,
CHAN RASTER in?, out!, BARRIER sync)

  RASTER raster  :
  CORDINATE cm0, cm1  :
  [WIDTH][WIDTH]INT trajectoire:
  --
  SEQ
    SEQ i=0 FOR WIDTH
      SEQ j=0 FOR WIDTH
        trajectoire[i][j] := COLOUR.WHITE

    WHILE TRUE
      SEQ
        in ? raster
        raster.clear(raster , COLOUR.WHITE)
        SEQ i=0 FOR WIDTH
          SEQ j=0 FOR WIDTH
            raster[i][j] := trajectoire[i][j]
        PAR
          chan0?  cm0
          chan1?  cm1

        fill.circle((INT ROUND cm0[x]),(INT ROUND cm0[y]),20 ,COLOUR.BLUE,raster)
```

```
        fill.circle((INT ROUND cm1[x]),(INT ROUND cm1[y]),20 ,COLOUR.GREEN ,raster)
        trajectoire[(INT ROUND cm1[y])] [(INT ROUND cm1[x])] := COLOUR.GREEN
        trajectoire[(INT ROUND cm0[y])] [(INT ROUND cm0[x])] := COLOUR.BLUE
        --

        out! raster
        SYNC sync
:

PROC main (CHAN BYTE kbd?, scr!)
  CHAN RASTER raster.in, raster.out       :
  CHAN CORDINATE body.b10, body.b01       :
  CHAN CORDINATE coord0,coord1            :
  BARRIER synchro                         :
  --
  CORDINATE cm0, cm1        :
  SPEED     vm0, vm1        :
  REAL64    mass0, mass1    :
  --
  INITIAL REAL64    delta  IS 0.08        :
  SEQ
    --  valeurs initiales
    erase.screen(scr!)
    cursor.x.y(2,2, scr!)
    out.string("Probleme des deux corps", 0 , scr!)
    --
    print.str(2,4,"coordonnee X du premier corp : ", scr!)
    read.real64(cm0[x],kbd?, scr!)
    print.str(2,5,"coordonnee Y du premier corp : ", scr!)
    read.real64(cm0[y],kbd?, scr!)
    print.str(2,6,"composante VX de la vitesse  du premier corp : ", scr!)
    read.real64(vm0[vx],kbd?, scr!)
    print.str(2,7,"composante VY de la vitesse  du premier corp : ", scr!)
    read.real64(vm0[vy],kbd?, scr!)
    print.str(2,8,"masse du premier corp : ", scr!)
    read.real64(mass0, kbd?, scr!)
    --
    print.str(2,10,"coordonnee X du deuxieme corp : ", scr!)
    read.real64(cm1[x],kbd?, scr!)
    print.str(2,11,"coordonnee Y du deuxieme corp : ", scr!)
    read.real64(cm1[y],kbd?, scr!)
    print.str(2,12,"composante VX de la vitesse  du deuxieme corp : ", scr!)
    read.real64(vm1[vx],kbd?, scr!)
    print.str(2,13,"composante VY de la vitesse  du deuxieme corp : ", scr!)
    read.real64(vm1[vy],kbd?, scr!)
    print.str(2,14,"masse du deuxieme corp : ", scr!)
    read.real64(mass1, kbd?, scr!)

    PAR
      raster.display.simple ("Probleme des deux corps", WIDTH, WIDTH, 1, raster.out?, raster.in!)
      PAR BARRIER synchro
        draw (coord0?, coord1?,mass0,mass1, raster.in?, raster.out!, synchro )
        body(body.b10?, body.b01!, cm0,vm0 , mass1,delta, coord0!, synchro )
        body(body.b01?, body.b10!, cm1,vm1 , mass0,delta, coord1!, synchro )
```

:

Un exemple ou deux corps de même masse, tournent chacun autour de l'autre en décrivant chacun une ellipse .

```
Probleme des deux corps

coordonnee X du premier corp : 200.0
coordonnee Y du premier corp : 400.0
composante VX de la vitesse  du premier corp : 0.0
composante VY de la vitesse  du premier corp : 1.0
masse du premier corp : 200000.0

coordonnee X du deuxieme corp : 600.0
coordonnee Y du deuxieme corp : 400.0
composante VX de la vitesse  du deuxieme corp : 0.0
composante VY de la vitesse  du deuxieme corp : -1.0
masse du deuxieme corp : 200000.0
```

La figure 16.2 correspond au cas de figure identique à celui qui est présenté mais où le deuxième corps a une masse de 100000.0.
Si le deuxième corps a une masse de 50.0,très faible par rapport à celle du premier, il tourne autour de ce dernier (fixe) en décrivant une ellipse.

16.3.1 Mise sur orbite d'un satellite

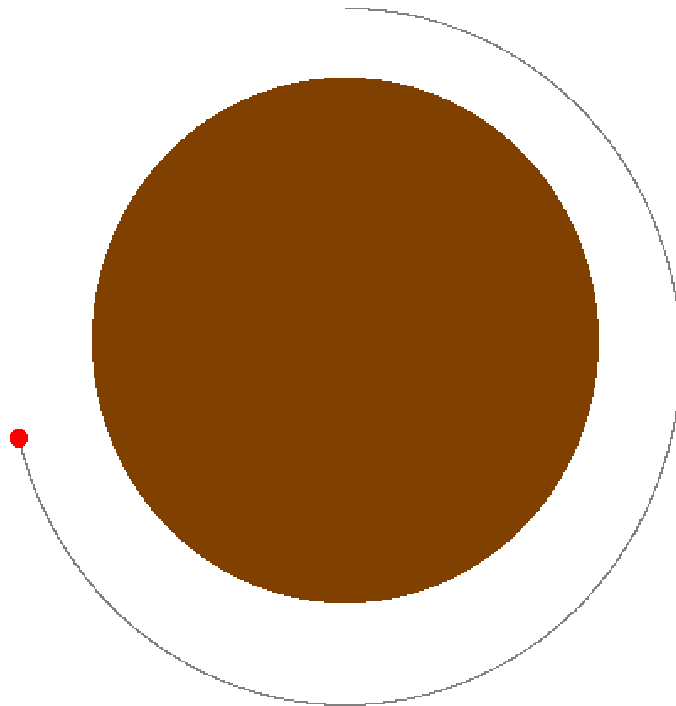


FIGURE 16.3 – Mise sur orbite

Cet exemple est un cas particulier du problème des deux corps où le second corps (le satellite) a une masse négligeable vis à vis de celle du premier.
Le second corps , plaçé à une altitude donnée , est animé initialement d'une vitesse VX suivant l'axe des X . Si VX est faible le corps chutte . Pour VX $>=$ 3.9 il se met en orbite . Si VX est trop important il y a évasion.

Le programme conserve le fichier gravite.occ vu précedemment. Le processus draw() est légèrement modifié pour tenir compte de ce que le corps massif est considéré comme fixe. Ces modifications sont apportées dans le fichier satel.occ ci dessous :

```
-- ch16_6.occ
--
-- satel.occ

#INCLUDE "gravite.occ"

#INCLUDE "sdlraster.module"
#INCLUDE "rastergraphics.module"


PROC draw (CHAN CORDINATE chan0?, chan1?,VAL REAL64 m0,m1,
CHAN RASTER in?, out!, BARRIER sync)
```

```
  RASTER raster       :
  CORDINATE cm0, cm1  :
  [WIDTH][WIDTH]INT trajectoire:
  --
  SEQ
    SEQ i=0 FOR WIDTH
      SEQ j=0 FOR WIDTH
        trajectoire[i][j] := COLOUR.WHITE

    WHILE TRUE
      SEQ
        in ? raster
        raster.clear(raster , COLOUR.WHITE)
        SEQ i=0 FOR WIDTH
          SEQ j=0 FOR WIDTH
            raster[i][j] := trajectoire[i][j]
        PAR
          chan0?  cm0
          chan1?  cm1

        fill.circle((INT ROUND cm0[x]),(INT ROUND cm0[y]),150 ,COLOUR.BROWN ,raster)
        IF
          raster[(INT ROUND cm1[y])][(INT ROUND cm1[x])] = COLOUR.BROWN
            -- crash !!
            STOP
          TRUE
            SEQ
              fill.circle((INT ROUND cm1[x]),(INT ROUND cm1[y]),5  , COLOUR.RED ,raster)
              trajectoire[(INT ROUND cm1[y])] [(INT ROUND cm1[x])] := COLOUR.GREY
        out! raster
        SYNC sync
:



PROC main (CHAN BYTE kbd?, scr!)
  CHAN RASTER raster.in, raster.out       :
  CHAN CORDINATE body.b10, body.b01       :
  CHAN CORDINATE coord0,coord1            :
  BARRIER synchro                         :
  --
  CORDINATE cm1        :
  SPEED     vm1        :
  REAL64    mass1      :
  INITIAL CORDINATE cm0 IS [400.0, 200.0] :
  INITIAL SPEED     vm0 IS [0.0, 0.0]     :
  INITIAL REAL64 mass0 IS 400000.0        :

  INITIAL CORDINATE cm1 IS [400.0, 10.0]  :
  INITIAL SPEED     vm1 IS [0.0, 0.0 ]    :
  INITIAL REAL64 mass1 IS 20.00           :
  --
  INITIAL REAL64    delta  IS 0.08        :
```

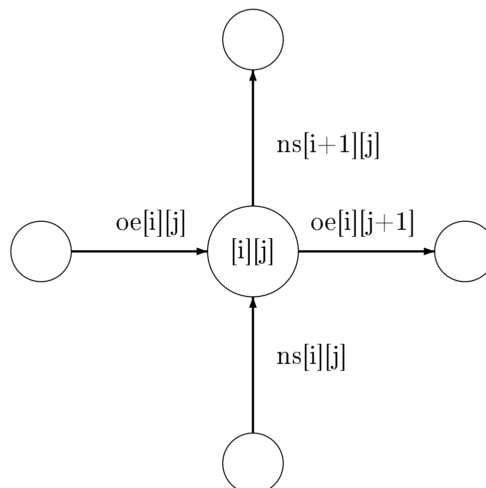
```
SEQ
  --  valeurs initiales
  erase.screen(scr!)
  cursor.x.y(0,2, scr!)
  out.string("Mise sur orbite", 0 , scr!)
  out.string("*nLa vitesse minimale de  mise en orbite est VX = 3.9",0,scr!)
  --

  print.str(2,8,"composante VX de la vitesse  du corp : ", scr!)
  read.real64(vm1[vx],kbd?, scr!)

  PAR
    raster.display.simple ("Mise sur orbite", WIDTH, WIDTH, 1, raster.out?, raster.in!)
    PAR BARRIER synchro
      draw (coord0?, coord1?,mass0,mass1, raster.in?, raster.out!, synchro )
      body(body.b10?, body.b01!, cm0,vm0 , mass1,delta, coord0!, synchro )
      body(body.b01?, body.b10!, cm1,vm1 , mass0,delta, coord1!, synchro )
:
```

Chapitre 17

La grille

17.1 La grille torique

Dans la grille torique à deux dimentions chaque ligne (resp chaque colonne) est constituée en un anneau.
Les anneaux reliant les lignes à colonne fixée sont identifiés sud.nord (ns).
Les anneaux reliant les colonnes à ligne fixée sont identifiés ouest.est(oe).



Les processeurs , situés à chaque noeud de la grille torique sont indexés comme une matrice 2x2.
Si proc(i,j) fait référence au processus situé sur la ligne i colonne j il posséde deux canaux entrants oe(i,j), ns(i,j) et deux canaux sortants ns(i+1,j) et oe(i,j+1). Les canaux entrants d'un processus ont même indexation que ce processus .
Pour les canaux sortants d'un processus la somme (i+1) dans l'écriture ns(i+1,j) est

calculée modulo le nombre de ligne et la somme (j+1) dans l'écriture oe(i, j+1) est calculée modulo le nombre de colonne.
La figure qui suit montre une grille torique 3x3 avec les indéxations de certains canaux ns[][] et oe[][].

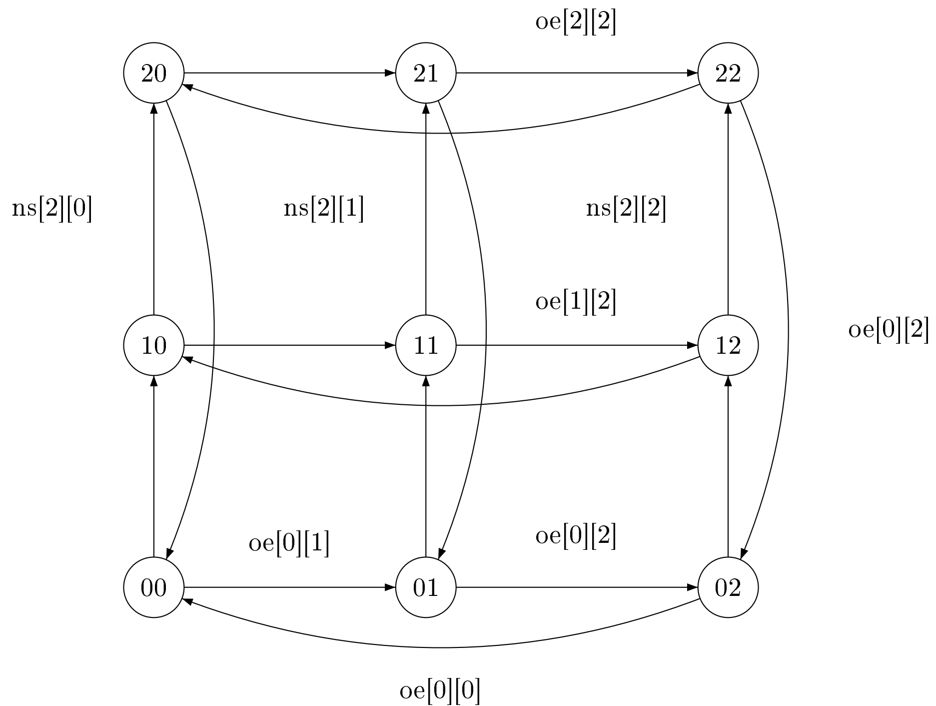


FIGURE 17.1 – Une grille torique 3x3

17.1.1 Le routage store and forward

Un flit est une unité d'information qui transite sur le réseau. Usuellement il est constitué d'une en-tête qui consigne le code de l'émetteur et celui du destinataire, un code de protocole, puis les données proprement dites. Un processeur situé en un noeud du réseau est constitué le plus souvent de deux processus qui s'exécutent en parallèle : un routeur et une unité de traitement.
Dans ce qui suit on s'intéresse au routage le plus simple : le routage store and forward. Dans ce type de routage toute l'information est concentrée dans un seul flit qui doit être stocké dans sa totalité sur chaque noeud par lequel ce flit transite.
Le protocole de routage :
Si le noeud proc(i,j) reçoit le flit par un canal sud.nord (i,j) et s'il n'est pas destinataire alors il le reémet sur le canal sud.nord(i+1,j).

Si le noeud proc(i,j) recoit le flit par le canal ouest.est(i,j) et s'il n'est pas destinataire :
si l'index de colonne j est celui du destinataire alors le flit est écrit dans le canal sud.nord(i+1,j)
sinon le flit est écrit dans le canal ouest.est(i, j+1).
Par exemple sur une grille 3x3 un flit qui est émis par le noeud (0,1) à destination du noeud (2,0) transitera par les noeuds (0,2),(0,0),(1,0), (2,0).

17.1.2 Le programme

```
-- ch17_1.occ
--
PROTOCOL STRING IS INT::[]BYTE      :
PROTOCOL PTCL IS BYTE; BYTE; STRING :

PROC process(VAL INT line,col,
             CHAN PTCL ns.out!, ns.in?, oe.out!, oe.in?,
             SHARED CHAN BYTE screen!)

:

PROC main(SHARED CHAN BYTE scr!)
  VAL INT lin IS 4  :
  VAL INT col IS 4  :
  [lin][col]CHAN PTCL nord.sud  :
  [lin][col]CHAN PTCL ouest.est :
  PAR i=0 FOR lin
    PAR j=0 FOR col
      process(i,j,nord.sud [(i+1)REM lin][j]!, nord.sud [i][j]?,
                  ouest.est[i][(j+1)REM col]!, ouest.est[i][j]?,
                  scr!)
:
```

Le protocole PTCL comprend :
le code sur un BYTE du processus émetteur. Les bits 0..3 codent la ligne et les bits 7..4 codent la colonne.
le code sur un BYTE du processus récepteur. Les bits 0..3 codent la ligne et les bits 7..4 codent la colonne.
la longueur du message codée par un INT.
le message constitué d'un tableau de caractères.
Un processus est caractérisé par sa ligne line et sa colonne col codés sur un INT de même que par ses canaux entrants ns.in ? , oe.in ?, et ses canaux sortants ns.out !, oe.out !.
Le processus main() déclare les canaux et lance les processus en parallèle.On remarque que les canaux sont bien indexés en accord avec les conventions énoncées au paragraphe 17.1.

17.1.3 La structure d'un processus

Un processus process() est constitué d'un routeur router() et d'une unité de traitement uc() reliés par un canal rout.uc de protocole PTCL. Le routeur a accés aux canaux d'entrées sorties. Il communique avec l'unité de traitement via le canal rout.uc. L'unité

de traitement procède à l'affichage des informations issues du routeur.
Qu'il soit émetteur ou non un processus scrute ses entrées sorties, reçoit ou route des informations et se termine. Si, au bout d'une seconde environ, aucune information n'a été lue sur les canaux d'entrée un chien de garde associé au routeur déclanche sa terminaison.

```
PROC process(VAL INT line,col,
             CHAN PTCL ns.out!, ns.in?, oe.out!, oe.in?,
             SHARED CHAN BYTE screen!)

  PROC router(VAL INT li, co,
              CHAN PTCL ns.o! ,ns.i?, oe.o!, oe.i?,
              CHAN PTCL info!)

  :
  -- router()

  PROC uc(CHAN PTCL info?, SHARED CHAN BYTE ecran!)

  :
  -- uc()

  CHAN PTCL rout.uc  :
  PAR
    router(line,col, ns.out!,ns.in?,oe.out!,oe.in?, rout.uc!)
    uc (rout.uc?, screen!)
:
-- process()
```

17.1.4 La structure du routeur

Le routeur est un processus séquentiel constitué de 3 processus :
Un processus émetteur défini par sa ligne et sa colonne. Par souci de simplicité la ligne et la colonne sont fixés à l'avance.Un seul processus est émetteur.
Un processus de routage proprement dit.
Un processus de communication avec l'unité de traitement.

Le processus de routage est un processus alternatif prioritaire composé de trois gardes :

```
PRI ALT
  ns.i? emet;recoit;len::buffer
    SEQ
      traite les informations issues du canal sud.nord
  oe.i? emet;recoit;len::buffer
    SEQ
      traite les informations issues du canal ouest.est
  clock? now1
    SEQ
      met en oeuvre le chien de garde
```

17.1.5 Le programme complet

```
-- ch17_2.occ
--
#USE "course.lib"

PROTOCOL STRING IS INT::[]BYTE      :
PROTOCOL PTCL IS BYTE; BYTE; STRING :



PROC process(VAL INT line,col,
             CHAN PTCL ns.out!, ns.in?, oe.out!, oe.in?,
             SHARED CHAN BYTE screen!)
  PROC router(VAL INT li, co,
              CHAN PTCL ns.o! ,ns.i?, oe.o!, oe.i?,
              CHAN PTCL info!)
    --
    VAL INT delai IS 1000000    :
    VAL []BYTE message IS "Hello World ..."  :
    BYTE emet , recoit, proc.id :
    INT len , now0, now1        :
    [256]BYTE buffer            :
    TIMER clock                 :
    INITIAL BOOL encore IS TRUE :
    INITIAL BOOL recu IS FALSE  :
    --
    -- partie emission L'emetteur est le processus (0,0)
    -- le processus recepteur est le processus (3,2)
    SEQ
      clock? now0
      proc.id := (BYTE li) + ((BYTE co)<< 4)
      IF
        (li=0) AND (co=0)
          SEQ
            emet   := #00
            recoit := #23
            IF
              (recoit >>4 ) = (BYTE co)
                ns.o! emet; recoit; (SIZE message)::message
```

```
        TRUE
          oe.o! emet; recoit; (SIZE message)::message
  TRUE
    SKIP
--
-- partie routage
WHILE encore
  PRI ALT
    ns.i? emet;recoit;len::buffer
      SEQ
        IF
          recoit= proc.id
            SEQ
              -- transmet a uc
              recu := TRUE
              info! emet;recoit;len::buffer
          TRUE
            ns.o! emet; recoit; len::buffer
        encore := FALSE

    oe.i? emet;recoit;len::buffer
      SEQ
        IF
          recoit= proc.id
            SEQ
              -- transmet a uc
              recu := TRUE
              info! emet;recoit;len::buffer
          TRUE
            BYTE dest.colon :
            SEQ
              dest.colon := recoit >> 4
              IF
                dest.colon = (BYTE co)
                  ns.o! emet; recoit; len::buffer
                TRUE
                  oe.o! emet; recoit; len::buffer
        encore := FALSE

    clock? now1
      IF
        (now1-now0) > delai
          encore := FALSE
        TRUE
          SKIP
--
-- transfert d'information vers uc()
-- dans le cas ou le processus n'est
-- pas destinataire . Le flit 0;0;1::"."
-- est arbitraire et n'est pas traite
IF
  NOT recu
    info! 0;0;1::"."
  TRUE
```

```
        SKIP
  :
  --  router()

  PROC uc(CHAN PTCL info?, SHARED CHAN BYTE ecran!)
    [256]BYTE buffer    :
    INT len             :
    BYTE emet, recoit   :
    BYTE lin.emet, col.emet     :
    BYTE lin.recoit,col.recoit  :
    SEQ
      info? emet;recoit;len::buffer
      lin.emet   := (emet BITAND #0F) + '0'
      col.emet   := ((emet BITAND #F0) >> 4) + '0'
      lin.recoit := (recoit BITAND #0F) + '0'
      col.recoit := ((recoit BITAND #F0) >> 4) + '0'
      IF
        len > 1
          CLAIM ecran!
            SEQ
              ecran! lin.emet
              ecran! col.emet
              out.string(" a emis: ",0, ecran!)
              SEQ i=0 FOR len
                ecran! buffer[i]
              out.string(" a : ",0,ecran!)
              ecran! lin.recoit
              ecran! col.recoit
              ecran! '*n'
        TRUE
          SKIP
  :
  --  work()

  CHAN PTCL rout.uc  :
  PAR
    router(line,col, ns.out!,ns.in?,oe.out!,oe.in?, rout.uc!)
    uc(rout.uc?, screen!)
:

PROC main(SHARED CHAN BYTE scr!)
  VAL INT lin IS 4  :
  VAL INT col IS 4  :
  [lin][col]CHAN PTCL nord.sud  :
  [lin][col]CHAN PTCL ouest.est :
  PAR i=0 FOR lin
    PAR j=0 FOR col
      process(i,j,nord.sud [(i+1)REM lin][j]!, nord.sud [i][j]?,
                  ouest.est[i][(j+1)REM col]!, ouest.est[i][j]?,
                  scr!)
:
```

Le processus émetteur est toujours (0,0) pour simplifier.Le processus récepteur est

(3,2). L'affichage sur le terminal est :

```
00 a emis: Hello World ... a : 32
```

17.1.6 Utiliser le parallélisme des liens

Dans ce type de routage on exploite le parallélisme des liens . Le message est décomposé en deux flits qui sont émis en parallèle en mode store and forward sur les canaux sortants sud.nord et ouest.est.
L'algorithme de routage store and forward est conservé sur chaque chemin. On notera que ces chemins sont arêtes disjoints.
Trois cas de figures sont à considérer en un noeud donné :

```
1 . Si le noeud est destinataire le processus en ce noeud  lit dans chaque
direction Sud.Nord et Ouest.Est.

2. Si le noeud n'est sur aucun des deux chemins le processus en ce noeud se
                termine par un time out dicte par le chien de garde.

3. Si le noeud est sur un des deux chemins le processus en ce noeud re route
 une fois le flit:

  Si le flit arrive par un canal Sud.Nord
       Si le noeud est sur la ligne du destinataire le flit est
                                   ecrit sur le canal Ouest.Est
       Sinon  le flit est          ecrit sur le canal Sud.Nord
  Si le flit arrive par le canal Ouest.Est:
       Si le noeud est sur la colonne du destinataire le flit est
                                    ecrit sur le canal Sud.Nord
       Sinon  le flit est           ecrit sur le canal Ouest.Est
```

Le programme de test :

```
-- ch17_3.occ
--
#USE "course.lib"

PROTOCOL STRING IS INT::[]BYTE      :
PROTOCOL PTCL IS BYTE; BYTE; STRING :


PROC process(VAL INT line,col,
             CHAN PTCL ns.out!, ns.in?, oe.out!, oe.in?,
             SHARED CHAN BYTE screen!)
  PROC router(VAL INT li, co,
              CHAN PTCL ns.o! ,ns.i?, oe.o!, oe.i?,
              CHAN PTCL info!)
    --
    VAL INT delai IS 1000000    :
    VAL []BYTE message1 IS " Emission Ouest.Est " :
    VAL []BYTE message2 IS " Emission Sud.Nord "  :
```

```
BYTE emet , recoit, proc.id :
INT len , now0, now1        :
[256]BYTE buffer            :
TIMER clock                 :
INITIAL BOOL encore IS TRUE :
INITIAL BOOL recu IS FALSE  :
INITIAL INT  nb.flits IS 2  :
SEQ
  clock? now0
  proc.id := (BYTE li) + ((BYTE co)<< 4)
  --
  --  emission par le processus (0,0)
  --
  IF
    (li=0) AND (co=0)
      SEQ
        emet   := #00
        recoit := #23
        PAR
          ns.o! emet; recoit; (SIZE message2)::message2
          oe.o! emet; recoit; (SIZE message1)::message1
    TRUE
      SKIP
  --
  --  routage proprement dit
  --
  WHILE encore
    PRI ALT
      ns.i? emet;recoit;len::buffer
        SEQ
          IF
            recoit= proc.id
              SEQ
                recu := TRUE
                nb.flits := nb.flits -1
                info! emet;recoit;len::buffer
                IF
                  nb.flits = 0
                    encore := FALSE
            TRUE
              SEQ
                -- teste si sur ligne de destination
                IF
                  (recoit BITAND #0F) = (BYTE li)
                    oe.o! emet; recoit;len::buffer
                  TRUE
                    ns.o! emet; recoit; len::buffer
                encore := FALSE

      oe.i? emet;recoit;len::buffer
        SEQ
          IF
            recoit= proc.id
              SEQ
```

```
                  recu := TRUE
                  nb.flits := nb.flits -1
                  info! emet;recoit;len::buffer
                  IF
                    nb.flits = 0
                      encore := FALSE
                    TRUE
                      SKIP
              TRUE
                SEQ
                  -- teste si sur colonne destination
                  IF
                    (recoit >> 4) = (BYTE co)
                      ns.o! emet; recoit; len::buffer
                    TRUE
                      oe.o! emet; recoit; len::buffer
                  encore := FALSE

        clock? now1
          IF
            (now1-now0) > delai
              encore := FALSE
            TRUE
              SKIP
    IF
      NOT recu
        SEQ
          info! 0;0;1::"."
          info! 0;0;1::"."
      TRUE
        SKIP
:
--  router()

PROC uc(CHAN PTCL info?, SHARED CHAN BYTE ecran!)
  [20]BYTE buffer1           :
  [20]BYTE buffer2           :
  INT len1, len2             :
  BYTE emet, recoit          :
  BYTE lin.emet, col.emet    :
  BYTE lin.recoit,col.recoit  :
  SEQ
    info? emet;recoit;len1::buffer1
    info? emet;recoit;len2::buffer2

    lin.emet   := (emet BITAND #0F) + '0'
    col.emet   := (emet  >> 4) + '0'
    lin.recoit := (recoit BITAND #0F) + '0'
    col.recoit := (recoit >> 4) + '0'
    IF
      len1 > 1
        CLAIM ecran!
          SEQ
            ecran! lin.emet
```

```
              ecran! col.emet
              out.string(" a emis: ",0, ecran!)
              SEQ i=0 FOR len1
                ecran! buffer1[i]
              ecran! ':'
              SEQ i=0 FOR len2
                ecran! buffer2[i]
              out.string(" a : ",0,ecran!)
              ecran! lin.recoit
              ecran! col.recoit
              ecran! '*n'
        TRUE
          SKIP
  :
  --  uc()

  CHAN PTCL rout.uc  :
  PAR
    router(line,col, ns.out!,ns.in?,oe.out!,oe.in?, rout.uc!)
    uc(rout.uc?, screen!)
:

PROC main(SHARED CHAN BYTE scr!)
  VAL INT lin IS 4  :
  VAL INT col IS 4  :
  [lin][col]CHAN PTCL nord.sud  :
  [lin][col]CHAN PTCL ouest.est :
  PAR i=0 FOR lin
    PAR j=0 FOR col
      process(i,j,nord.sud [(i+1)REM lin][j]!, nord.sud [i][j]?,
                  ouest.est[i][(j+1)REM col]!, ouest.est[i][j]?,
                  scr!)
:
```

17.2 Un programme de trie sur la grille

Ce programme de trie sur la grille bien que n'étant pas optimal, (voir à ce sujet : [Lei95] page 17) constitue un bon exemple d'algorithme à grain fin, dans lequel les processus impliqués n'effectuent que des taches élémentaires.

17.2.1 Le principe de l'algorithme

L'algorithme repose sur la remarque suivante :
Soient deux entiers positifs x et y codés sur n bits.
$x = (x_{n-1}, \ldots, x_1, x_0)$, $y = (y_{n-1}, \ldots, y_1, y_0)$ alors :
$x < y$ ssi $\exists i : 0 \leq i < n$ tel que : $x_i = 0, y_i = 1$ et : $\forall k : i < k$ alors $x_k = y_k$

L'algorithme de principe qui découle de la remarque est illustré ci dessous où deux nombres 4 et 3, codés sur 4 bits, sont comparés. Les deux bits de fort poids sont comparés en même temps qu'une commande initiale '=' est associée à cette comparaison . Les deux bits étant égaux aucune action n'est entreprise et l'on passe à la comparaison des 2 bits suivants (bits 2) qui eux, sont différents.

La commande '>' est générée en même temps que les bits sont permuttés.
Cette commande est propagée lors des cycles suivants et elle conduit à la permutation de tous les couples de bits de même rang jusqu'aux bits de poids faible de rang 0 .

```
= 0 0     0 0     0 0     0 0     0 0
  1 0   > 1 0     0 1     0 1     0 1
  0 1     0 1   > 0 1     1 0     1 0
  0 1     0 1     0 1   > 0 1     1 0
```

Mise en oeuvre.

Les nb.data nombres codés sur nb.bits à trier sont au préalable mis en forme dans un tableau de (nb.data + nb.bits-1) colonnes sur nb.bits lignes. Les bits de poids fort sont sur la ligne 0 du tableau et les bits suivants décalés vers la gauche d'une colonne d'une ligne à l'autre. Les points symbolisent une absence de valeur. Pour les deux nombres 3 et 4 cela donne :

```
. . . 0 0
. . 1 0 .
. 0 1 . .
0 1 . . .
```

Les colonnes de ce tableau, à commencer par la dernière sont injectées une à une sur une grille de processus de dimensions (nb.data + nb.bits-1) colonnes et nb.bits lignes dont les valeurs initiales valent '.' .

```
0  ->  . . . . .      0 . . . .
.  ->  . . . . .      . . . . .
.  ->  . . . . .      . . . . .
.  ->  . . . . .      . . . . .
```

phase 1. commande =

```
0  ->  0 . . . .      0 0 . . .
0  ->  . . . . .      0 . . . .
.  ->  . . . . .      . . . . .
.  ->  . . . . .      . . . . .
```

phase 2. commande =

```
.  ->  0 0 . . .      . 0 0 . .
1  ->  0 . . . .      0 1 . . .
1  ->  . . . . .      1 . . . .
.  ->  . . . . .      . . . . .
```

phase 3. commande > est propagée.le bit 2 de 4 passe devant le bit 2 de 3.

```
.  ->  . . . . .      . . 0 0 .
.  ->  . . . . .      . 0 1 . .
0  ->  . . . . .      1 0 . . .
1  ->  . . . . .      1 . . . .
```

phase 4. la commande $>$ est propagée . le bit 1 de 4 passe devant le bit 1 de 3.

```
.  ->  . . 0 0 .        . . . 0 0
.  ->  . 0 1 . .        . . 0 1 .
.  ->  1 0 . . .        . 1 0 . .
0  ->  1 . . . .        1 0 . . .
```

phase 5. La commande $>$ est propagée. le bit 0 de 4 passe devant le bit 0 de 3.

Le lecteur est invité à refaire cette analyse dans le cas où les valeurs à trier sont 3 et 4 . Comme $3 < 4$ une commande '$<$' aurait été générée et propagée.

17.2.2 Mise en oeuvre de l'algorithme

A l'issue de l'analyse qui vient d'être faite on voit que l'algorithme met en oeuvre deux processus qui s'exécutent en parallèle : un processus injecteur et une grille de processus reliés par nb.bits canaux. A chaque phase, dans la progression de l'algorithme, les

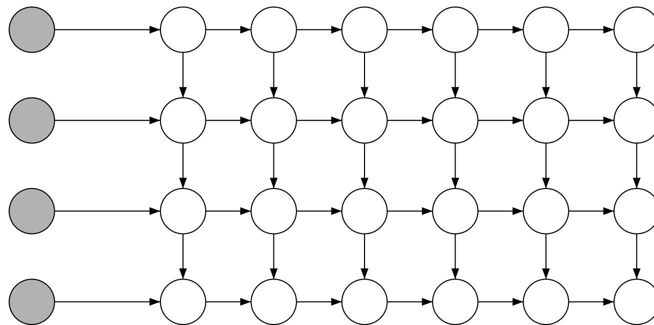


FIGURE 17.2 – L'injecteur et la grille

données triées progressent d'ouest en est le long des canaux c.oe. En même temps, les commandes transitent du nord vers le sud le long des canaux c.ns.
Pour la commodité de la programmation l'injecteur et la grille sont vus comme une grille unique dont la colonne 0 est dédiée à l'injecteur.

Les actions effectuées par un processus sur la grille.

Soit un processus de coordonnées (i,j) $j > 0$ sur la grille .
Il reçoit une valeur vin sur le canal c.oe[j-1][i].
Les valeurs possibles de ces dernières sont '.', '0','1' et le $ utilisé pour le terminaison propre de l'ensemble des processus.
Il reçoit une commande cin sur le canal c.ns[i-1][j].
Les valeurs possibles de ces dernières sont EQUAL (notre '=') , LEFT (notre '$>$') et RIGHT ('$<$').
Les valeurs vin et cin conjointement avec les valeurs de va (valeur initiale) et ca (commande initiale) déterminent les valeurs de :
vout : valeur transmise à la cellule de droite sur le canal c.oe[i][j].

cout : commande transmise à la cellule inférieure sur le canal c.ns[i][j].
Initialement va = '.' et ca = EQUAL .

```
CASE ca
  EQUAL
    IF
      v.in > va
        SEQ
          v.out := v.in
          c.out := LEFT
          -- va inchange
      v.in < va
        SEQ
           v.out := va
           c.out := RIGHT
           va := v.in
      v.in = va
        SEQ
          v.out := v.in
          c.out := EQUAL
          -- va inchange

  LEFT
    SEQ
      v.out := v.in
      c.out := LEFT
      -- va inchange

  RIGHT
    SEQ
      v.out := va
      c.out := RIGHT
      va := v.in
```

On remarque que si cin vaut LEFT où RIGHT cette commande est propagée et donc cout := cin.
Chaque processus sur la grille après lecture de vin et cin exécute p.unit(vin, ca, va) qui retourne vout, cout et va.. ca est toujouts egal à cin.
Une exécution typique est :

```
SEQ
  PAR
    coe.in? vin
    cns.in? cin
  vout, cout, va := p.unit(vin, ca,va)
  PAR
    coe.out! vout
    cns.out! cout
  ca := cin
```

Finalement, par soucis pédagogique mais aussi pour rendre plus claire l'indexation des canaux, les processus de la grille ont été regroupés en 6 grandes zones correspondant aux diverses configurations de canaux entrants où sortants.
Les deux processus sup.est() et inf.est() correspondant à la ligne 0 (resp nbits-1) de la dernière colonne de la grille.
bord.nord() qui regroupe tous les processus sur la ligne 0 à l'exception de sup.est().
bord.sud() qui regroupe tous les processus sur la ligne nbits-1 à l'exception de inf.est().

bord.est() qui regroupe tous les processus sur la dernière colonne à l'exception de sup.est() et inf.est().
interne() qui regroupe tous les autres processus .

```
-- ch17_4.occ
--
#USE "course.lib"

PROC affiche(VAL INT line, colon,VAL BYTE va, SHARED CHAN BYTE out!)
  CLAIM out!
    SEQ
      flush(out!)
      cursor.x.y((BYTE colon),(BYTE line)+1, out!)
      out! va
:

BYTE,BYTE, BYTE  FUNCTION  p.unit (VAL BYTE v.in,  ca, va)
  --
  -- retourne v.out, c.out, nva
  --
  VAL BYTE EQUAL IS 1 :
  VAL BYTE LEFT  IS 2 :
  VAL BYTE RIGHT IS 4 :

  BYTE v.out , c.out  :
  BYTE nva            :
  VALOF
    IF
      v.in = '.'
        IF
          va = '.'
            SEQ
              v.out := va
              c.out := EQUAL
              nva := va
          va <> '.'
            SEQ
              v.out := '.'
              c.out := LEFT
              nva := va
      v.in = '$'
        SEQ
          v.out := '$'
          c.out := LEFT
          nva := va

      --v.in vaut 0 ou 1
      TRUE
        IF
          va = '.'
            SEQ
              v.out := va
              c.out := RIGHT
              nva := v.in
```

```
          va <> '.'
            IF
              ca = EQUAL
                IF
                  v.in > va
                    SEQ
                      v.out := v.in
                      c.out := LEFT
                      nva := va
                  v.in < va
                    SEQ
                      v.out := va
                      c.out := RIGHT
                      nva := v.in
                  v.in = va
                    SEQ
                      v.out := v.in
                      c.out := EQUAL
                      nva := va
              ca = LEFT
                SEQ
                  v.out := v.in
                  c.out := LEFT
                  nva := va
              ca = RIGHT
                SEQ
                  v.out := va
                  c.out := RIGHT
                  nva := v.in
    RESULT v.out, c.out, nva
:

--  ************************************************************


PROC sup.est (VAL INT col, CHAN BYTE coe.in?, cns.out!, SHARED CHAN BYTE out!)
  BYTE vin              :
  BYTE vout, cout       :
  BYTE ca, va           :
  VAL BYTE EQUAL IS 1   :
  INITIAL BOOL encore IS TRUE :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        coe.in? vin
        vout, cout, va := p.unit(vin, ca,va)
        cns.out! cout
        IF
          vin = '$'
            SEQ
              encore := FALSE
```

```
              CLAIM out!
                SEQ
                  flush(out!)
                  cursor.x.y((BYTE col),0, out!)
                  out! va
          TRUE
            SKIP
:


PROC inf.est(VAL INT col, CHAN BYTE coe.in?,cns.in?, SHARED CHAN BYTE out!)
  BYTE vin , cin        :
  BYTE vout, cout       :
  BYTE ca, va           :
  VAL BYTE EQUAL IS 1   :
  INITIAL BOOL encore IS TRUE :
  INITIAL BYTE colon IS 1  :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        PAR
          coe.in? vin
          cns.in? cin
        vout, cout, va := p.unit(vin, ca,va)
        ca := cin
        affiche(3,col,va,out!)
        IF
          vin = '$'
            encore := FALSE

          TRUE
            SKIP
:

PROC bord.nord (VAL INT col, CHAN BYTE coe.in?, coe.out!,cns.out!, SHARED CHAN BYTE out!)
  BYTE vin              :
  BYTE vout, cout       :
  BYTE ca, va           :
  VAL BYTE EQUAL IS 1   :
  INITIAL BOOL encore IS TRUE :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        coe.in? vin
        vout, cout, va := p.unit(vin, ca,va)
        PAR
          coe.out! vout
          cns.out! cout
```

```
        affiche(0,col,va,out!)
        IF
          vin = '$'
            encore := FALSE
          TRUE
            SKIP
:


PROC bord.sud (VAL INT col,  CHAN BYTE coe.in?,cns.in?, coe.out!, SHARED CHAN BYTE out!)
  VAL BYTE EQUAL IS 1   :
  BYTE vin , cin        :
  BYTE vout, cout       :
  BYTE ca, va           :
  INITIAL BOOL encore IS TRUE :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        PAR
          coe.in? vin
          cns.in? cin
        vout, cout, va := p.unit(vin, ca,va)
        coe.out! vout
        ca := cin
        affiche(3,col,va,out!)
        IF
          vin = '$'
            encore := FALSE
          TRUE
            SKIP
:

PROC bord.est(VAL INT li,col, CHAN BYTE coe.in?, cns.in?, cns.out!,  SHARED CHAN BYTE out!)
  BYTE vin , cin        :
  BYTE vout, cout       :
  BYTE ca, va           :
  VAL BYTE EQUAL IS 1 :
  INITIAL BOOL encore IS TRUE :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        PAR
          coe.in? vin
          cns.in? cin
        vout, cout, va := p.unit(vin, ca,va)
        cns.out! cout
        ca := cin
        affiche(li,col,va,out!)
```

```
        IF
          vin = '$'
            encore := FALSE
          TRUE
            SKIP

:


PROC interne(VAL INT li,co,  CHAN BYTE coe.in?,cns.in?, coe.out!,cns.out!, SHARED CHAN BYTE out
  BYTE vin , cin        :
  BYTE vout, cout       :
  BYTE ca, va           :
  VAL BYTE EQUAL IS 1   :
  INITIAL BOOL encore IS TRUE :
  SEQ
    ca := EQUAL
    va := '.'
    vin := 0
    WHILE encore
      SEQ
        PAR
          coe.in? vin
          cns.in? cin
        vout, cout, va := p.unit(vin, ca,va)
        PAR
          coe.out! vout
          cns.out! cout
        ca := cin
        affiche(li,co,va,out!)
        IF
          vin = '$'
            SEQ
              encore := FALSE
          TRUE
            SKIP
:

--  ////////////////////////////////////////////////////////////

PROC main(CHAN BYTE kbd?, SHARED CHAN BYTE scr!)
  VAL []BYTE data IS [4,3,10,6,11,5,5,0,13, 8]  :
  VAL INT n.data IS 10                          :
  VAL INT nbit IS 4                             :
  VAL INT nco IS (n.data  + (nbit-1))           :
  VAL INT max.data IS 100                       :
  VAL INT nb.col IS (n.data + nbit)             :
  [nbit][max.data]BYTE buffer                   :
  [nbit][ nco] CHAN BYTE c.oe                   :
  [nbit][ nco] CHAN BYTE c.ns                   :
  SEQ
    --
    --  preparation du tableau buffer[][] a injecter a partir de []data
    --  max.data : taille max de donnees a trier
```

```
--
SEQ i= 0 FOR nbit
  SEQ
    SEQ k=0 FOR i
      buffer[i][k] := '.'
    SEQ k=0 FOR n.data
      IF
        (data[k] BITAND (1 << ((nbit-1)-i))) > 0
          buffer[i][k+i] := '1'
        TRUE
          buffer[i][k+i] := '0'
    SEQ k=0 FOR (nbit-1)-i
      buffer[i][(nb.col-2)-k]:= '.'
SEQ i=0 FOR nbit
  buffer[i][nb.col-1] := '$'
--
--execution en parallele de l'injecteur et de la grille
--
SEQ
  CLAIM scr!
    SEQ
      erase.screen(scr!)
      flush(scr!)

  --  les processus de la grille

  PAR
    sup.est(nco-1, c.oe[0][nco-2]?,c.ns[0][nco-1]!, scr!)
    inf.est(nco-1, c.oe[nbit-1][nco-2]?, c.ns[nbit-2][nco-1]?, scr!)

    PAR i=1 FOR (nco-2)
      bord.nord(i, c.oe[0][i-1]?, c.oe[0][i]!,c.ns[0][i]!, scr!)

    PAR i=1 FOR nbit-2
      PAR j=1 FOR nco-2
        interne(i,j, c.oe[i][j-1]?,c.ns[i-1][j]?, c.oe[i][j]!, c.ns[i][j]!, scr!)

    PAR i=1 FOR (nco-2)
      bord.sud(i, c.oe[nbit-1][i-1]?,c.ns[nbit-2][i]?, c.oe[nbit-1][i]!, scr!)

    PAR i=1 FOR (nbit-2)
      bord.est(i,nco-1,c.oe[i][nco-2]?, c.ns[i-1][nco-1]?, c.ns[i][nco-1]!, scr!)

    --  l'injecteur

    BYTE byte:
    SEQ j=0 FOR nb.col
      SEQ
        PAR i=0 FOR nbit
          c.oe[i][0]! buffer[i][j]

  --  fin du trie

  CLAIM scr!
```

```
        SEQ
          flush(scr!)
          cursor.x.y(1,8, scr!)
          out.string("*n*nBY...*n",0, scr!)
:
```

A l'exécution l'écran affiche :

```
0000001111..
0011110001..
0100010110..
0101100011..

BY...
```

17.3 Le routage whorm hole

Dans ce type de routage, l'information est découpée en paquets où flits. Le flit de tête qui contient seul l'adresse de destination ouvre la route en progressant de noeud en noeud et monopolise pour le profit des flits qui suivent les ressources de routage. Le flit de queue libère le chemin emprunté au fur et à mesure de sa progression vers la destination. Le routage whorm hole est préféré au mode store and forward car les flits ne nécessitent pas de gros volumes de stockage (un noeud stocke un flit). L'inconvénient de ce mode de routage est de réquisitionner à son profit tous les noeuds adjacents occupés par des flits. De ce fait les risques d'interblocage sont grands si plusieurs messages sont en concurrence pour être transférés dans le réseau.
Supposons pour fixer les idées un anneau unidirectionnel de 10 processeurs indexés de 0 à 9. Le processeur 0 émet vers le 8 un message M0 et le processeur 6 émet vers le 3 un message M6 . Ces deux messages sont supposés contenir au moins 10 flits.
Le message M0 va réquisitionner à son profit les noeuds [0 .. 8] et le message M6 les noeuds 6,7,8,9,0,1,2,3 au fur et à mesure de leur progression.
Si on suppose que les messages progressent d'un noeud simultannement et démarrent en même temps alors quand M0 atteint le noeud 3 M6 atteint le noeud 9. M6 est bloqué mais M0 peut progresser jusqu'au noeud 5 et est bloqué à son tour. Pour remédier à ce genre de problème un canal réel donne naissance à plusieurs canaux virtuels qui sont implémantés par multiplexage.

17.3.1 Implantation des canaux virtuels

Pour illustrer l'implémantation des canaux virtuels en mode whorm hole nous prenons l'exemple simple d'un anneau de 20 noeuds. Chaque processus associé à un noeud est constitué de deux processus s'exécutant en parallèle : un routeur (router) et une unité centrale (uc). Le routeur et l'uc communiquent par deux canaux : l'un rout.uc du routeur vers l'uc et l'autre uc.rout de l'uc vers le router.

```
  CHAN WHORM rt.uc, uc.rt :
  PAR
    router(in?, out!, uc.rt?, rt.uc!)
```

```
    uc(index, emet, uc.rt!, rt.uc?, screen!)
:
```

Le routeur reçoit des flits en provenance de son prédécesseur sur l'anneau et de l'unite centrale. Les flits provenants du prédécesseur sont soit transmis au noeud qui lui succède soit transmis à l'uc.
L'uc peut émettre ET recevoir des flits. Si elle reçoit des flits elle a pour tache de concaténer les chaines de caractères portées par ces derniers. Chaque flit porte une chaine de 3 caractères pour simplifier.
La figure qui suit illustre la structure d'un noeud.

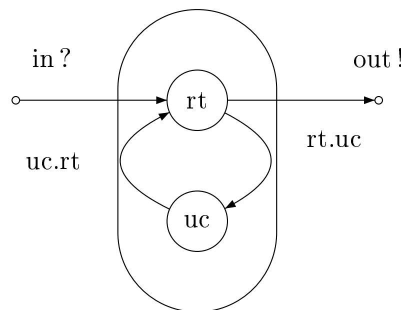


FIGURE 17.3 – Un noeud de l'anneau

17.3.2 Le protocole adopté

Le protocole (variable) WHORM est associé à la structure des flits.

```
DATA TYPE STRING IS [3]BYTE:

PROTOCOL PWHM0 IS INT ;INT ; STRING   :
PROTOCOL PWHM1 IS INT ; STRING        :
PROTOCOL WHORM
  CASE
    first ; PWHM0     --  flit de  tete
    inter ; PWHM1     --  flit intermediaire
    last  ; PWHM1     --  flit de queue
:
```

Le tag first est dédié à l'en-tête de protocole PWHM0 associé à l'adresse de destination, au numéro de canal virtuel et aux données.
Le tag inter est dédié à un flit quelconque de protocole PWHM1 associé au numero de canal virtuel et aux données.
Le tag last est dédié au flit de queue de protocole PWHM1 associé au numero de canal virtuel et aux données.

17.3.3 Le routage

L'unité centrale

En réception les flits sont concaténés au fur et à mesure de leur arrivée. Une fois le flit de queue reçu la chaine résultant de la concaténation des flits est affichée précédée du numéro du noeud récepteur.

En émission 3 flits sont émis : Un flit de tête, un flit interne et un flit de queue. Tous les flits sont porteurs d'un numéro de canal virtuel (indx) et d'une chaine de 3 BYTES. Le flit de tête porte en plus l'adresse du destinataire. L'index indx d'un noeud émetteur sert de numéro de canal virtuel. On peut donc gérer 20 canaux virtuels par canal comme nous le verrons dans l'application.

```
SEQ
   dest := (indx + 10) REM 20            -- emet a distance 10 modulo 20
   SEQ i=0 FOR 3
     data1[i] := 'A' + (BYTE indx)
   uc.out! first ;  dest ; indx ; data1  -- le numero de canal virtuel  := indx
   uc.out! inter ;  indx ; data1
   uc.out! last  ;  indx ; data1
```

Le routeur

La réception des flits provenant soit de l'uc soit de l'extérieur est gérée par un processus alternatif.

```
PRI ALT
  r.in? CASE
    -- traite les flits externes
  uc.in? CASE
    -- traite les flits provenant de l'uc
 clock? now1
    -- chien de garde
```

Une variable locale au routeur : virt.canal mémorise le numéro de canal virtuel porté par tous les flits destinés à l'actuel noeud . Initialement virt.canal vaut -1.(Les valeurs possibles de virt.canal sont les entiers de 0 A 19). Dans une version plus élaborée que ce programme il faudrait envisager que plusieurs noeuds émettent en parallèle des message à destination d'un noeud déterminé. Si le routeur reçoit des flits provenant de l'uc il les transmet tels quels à son successeur dans l'anneau.

Le routeur reçoit des flits en provenance de l'extérieur alors si :

Le flit est un flit de tête.
Si le destinataire est l'actuel noeud le flit est redirigé sur l'uc et le numéro de canal virtuel est mémorisé dans la variable virt.canal. sinon il est émis à vers le noeud suivant.

Le flit est un flit interne
Le numéro de canal est comparé à la valeur de la variable virt.canal. S'il y a égalité le flit est redirigé sur l'uc sinon il est émis à vers le noeud suivant.

Le flit est un flit de queue
Le numéro de canal est comparé à la valeur de la variable virt.canal. S'il y a égalité le flit est redirigé sur l'uc et la variable virt.canal est remise à -1, sinon il est émis vers le noeud suivant.

Le programme

Si la valeur du paramètre booléen emet de process() vaut TRUE ce processus est émetteur.
Dans ce programme tous les noeuds d'indice i sont émetteurs vers le noeud d'indice (i + 10) sauf le noeud 0 pour éviter un interblocage. Il en résulte qu'ils sont tous récepteurs sauf le noeud d'indice 10.

```
-- ch17_5.occ
--
#USE "course.lib"

DATA TYPE STRING IS [3]BYTE:

PROTOCOL PWHM0 IS INT ;INT ; STRING   :
PROTOCOL PWHM1 IS INT ; STRING        :
PROTOCOL WHORM
  CASE
    first ; PWHM0     --  flit de  tete
    inter ; PWHM1     --  flit intermediaire
    last  ; PWHM1     --  flit de queue
:


VAL INT delai IS 100000   :

PROC process(VAL INT index, VAL BOOL emet, CHAN WHORM in?, out!, SHARED CHAN BYTE screen!)

  PROC router(VAL INT indx, CHAN WHORM r.in?, r.out!, CHAN WHORM uc.in?, uc.out!)
    INT dest, emet             :
    INT now0, now1             :
    STRING data                :
    TIMER clock                :
    SEQ
      clock? now0
      INITIAL BOOL encore IS TRUE     :
      INITIAL INT virt.canal IS -1    :
      WHILE encore
        PRI ALT
          r.in? CASE
            first; dest ; emet; data
              IF
                dest = indx
                  SEQ
                    uc.out! first; dest ; emet; data
                    virt.canal := emet
                dest <> indx
                  r.out! first; dest ; emet; data
```

```
          inter; emet; data
            IF
              emet = virt.canal
                uc.out! inter; emet; data
              emet <> virt.canal
                r.out! inter; emet; data

          last; emet; data
            IF
              emet = virt.canal
                SEQ
                  uc.out! last; emet; data
                  virt.canal := -1
              emet <> virt.canal
                r.out! last; emet; data

        uc.in? CASE
          first; dest ; emet; data
            r.out! first; dest ; emet; data
          inter; emet; data
            r.out! inter; emet; data
          last; emet; data
            r.out! last; emet; data

        clock? now1
          SEQ
            IF
              (now1-now0) >=  delai
                encore := FALSE
              (now1-now0) < delai
                SKIP
:-- router()

PROC uc(VAL INT indx, VAL BOOL uc.emet, CHAN WHORM uc.out!, uc.in?,  SHARED CHAN BYTE u.scr!)

  PROC append(VAL INT debut, VAL STRING source, RESULT []BYTE dest)
    INT len.src :
    SEQ
      len.src := SIZE source
      SEQ i= 0 FOR len.src
        dest[debut + i] := source[i]
  :-- append()

  --    uc : partie emission

  PAR
    IF
      uc.emet = TRUE
        INT dest1     :
        STRING data1  :
        SEQ
          dest1 := (indx + 10) REM 20
          SEQ i=0 FOR 3
```

```
          data1[i] := 'A' + (BYTE indx)
        uc.out! first ;  dest1 ; indx ; data1
        uc.out! inter ;  indx ; data1
        uc.out! last  ;  indx ; data1

    uc.emet = FALSE
      SKIP

  --  uc : partie reception

  STRING data2        :
  INT dest2, emet     :
  INT now0, now1      :
  TIMER time          :
  SEQ
    time? now0
    INITIAL INT id IS 0         :
    INITIAL BOOL encore IS TRUE :
    INITIAL [20]BYTE str IS [i=0 FOR 20 | 0]     :
    WHILE encore
      PRI ALT
        uc.in? CASE
          first; dest2 ; emet; data2
            SEQ
              append(id, data2, str)
              id := id + 3

          inter; emet; data2
            SEQ
              append(id, data2, str)
              id := id + 3

          last; emet; data2
            SEQ
              append(id, data2, str)
              CLAIM u.scr!
                SEQ
                  flush(u.scr!)
                  u.scr! ' '
                  out.int(indx,0, u.scr!)
                  u.scr! ':'
                  SEQ i=0 FOR (SIZE str)
                    u.scr! str[i]
              encore := FALSE

        time? now1
          SEQ
            IF
              (now1-now0) >=  delai
                encore := FALSE
              (now1-now0) < delai
                SKIP
:-- uc()
```

```
  --
  --  process
  --

  CHAN WHORM rout.uc, uc.rout    :
  PAR
    router(index, in?, out!, uc.rout?, rout.uc!)
    uc(index, emet, uc.rout!, rout.uc?, screen!)

:-- process()


PROC main(SHARED CHAN BYTE scr!)
  [20]CHAN WHORM canal                           :
  INITIAL [20]BOOL emet IS [i=0 FOR 20 |TRUE ]  :
  SEQ
    emet[0] := FALSE
    PAR i= 0 FOR 20
      process(i,emet[i],  canal[i]?, canal[(i+1) REM 20]!, scr!)

    CLAIM scr!
      SEQ
        flush(scr!)
        out.string("*nBY ... *n",0, scr!)
:-- main()
```

Chapitre 18

L'hypercube

18.1 Généralités

L'hypercube d'ordre n noté $H(n)$ est défini récursivement par :
$H(0)$ possède un sommet codé sur 0 bits et possède 0 arêtes.
Les sommets de $H(n)$ sont codés sur n bits.
$H(n+1)$ est construit à partir de $H(n)$ en le dupliquant.
Si on note $(x_{n-1}, \ldots, x_0)$ le code binaire d'un sommet quelconque de $H(n)$ il donne naissance à deux sommets homologues dans $H(n+1)$.
Ces sommets sont notés : $(0, x_{n-1}, \ldots, x_0)$ et $(1, x_{n-1}, \ldots, x_0)$.
Les arêtes de chaque H(n) sont conservées et de nouvelles sont crées en reliant deux à deux leurs sommets homologues.
Il découle de la construction de $H(n)$ que :
Le nombre de sommets vaut : 2^n et le nombre d'arêtes vaut : $n.2^{n-1}$

Exemples d'hypercubes :
$H(1)$ posséde deux sommets codés 0 et 1 et 1 arête les reliant.
$H(2)$ posséde quatre sommets codés 00 et 01 , 10 , 11 et 4 arêtes.

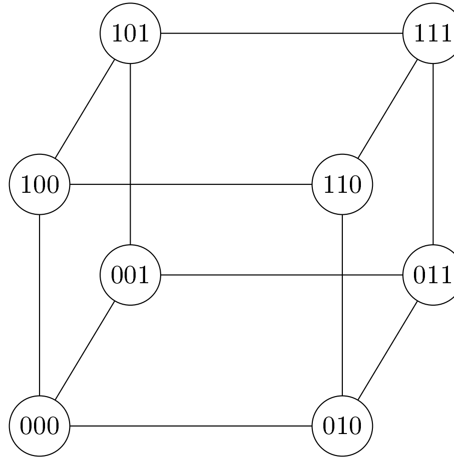


FIGURE 18.1 – Un hypercube d'ordre 3

Rappelons que le OU exclusif se note >< en Occam.
Si u, v sont deux entiers qui codent deux sommets de $H(n)$ leur distance $d(u, v)$ vaut : $d(u, v) = |u >< v|$. Dans cette notation $|w|$ dénote le nombre de bits à 1 dans l'écriture binaire de w. Comme deux sommets adjacents de $H(n)$ ont leur code qui diffère de 1 bit l'algorithme de routage dans $H(n)$ consiste , à chaque étape, à diminuer la distance entre l'actuel noeud et celui de destination en diminuant cette dernière de 1.

18.2 Le codage des canaux

Pour fixer les idées nous nous intéressons dans ce qui suit à $H(3)$ et nous noterons (i, j, k) le code d'un sommet où les indices i, j, k valent 0 ou 1.
On notera $\overline{i} = 0$ si $i = 1$ et $\overline{i}$ =1 si $i = 0$.
On notera c0(s) , c1(s), c2(s) chacun des 3 canaux sortants d'un sommet $s = (i, j, k)$ de $H(3)$ qui vérifient :
c0(s) relie (i, j, k) à $(i, j, \bar{k})$. Il relie donc un sommet s à un sommet adjacent qui ne diffère de s que par le bit 0.
c1(s) relie (i, j, k) à $(i, \bar{j}, k)$ .Il relie donc un sommet s à un sommet adjacent qui ne diffère de s que par le bit 1.
c2(s) relie (i, j, k) à $(\bar{i}, j, k)$ .Il relie donc un sommet s à un sommet adjacent qui ne diffère de s que par le bit 2.
Comme il y a 8 sommets dans $H(3)$ nous venons de définir 24 canaux (3 canaux par sommet). Nous savons par ailleurs que dans $H(3)$ il y a 12 = (3.2^2) arêtes. En fait à chaque canal sortant que nous venons de définir correspond un canal entrant si bien que les 24 canaux correspondent à deux arêtes d'orientations différentes .
Plus précisemment pour tout $s = (i, j, k)$ posons $T0(s) = (i, j, \bar{k})$.
Alors pour tout s au canal sortant c0(s) est associé le canal entrant c0($T0(s)$) si bien qu'une écriture dans c0(s) correspond à une lecture dans c0($T0(s)$).
De la même manière si s =(i, j, k) on définit $T1(s) = (i, \bar{j}, k)$ et $T2(s) = (\bar{i}, j, k)$ et ainsi a tout canal sortant c1(s) est associé un canal entrant c1($T1(s)$) et a tout canal sortant c2(s) est associé un canal entrant c2($T2(s)$). Ce qui donne la structure générale

suivante pour la connectique des processus dans $H(3)$:

```
-- ch18_1.occ
--
PROTOCOL STRING IS INT::[]BYTE    :
PROTOCOL PTCL IS INT; INT; STRING :

PROC process(VAL INT i, CHAN PTCL c0.out!,  c1.out!,  c2.out!,
                        c0.in?, c1.in?, c2.in?,SHARED CHAN BYTE screen!)
:

PROC main(SHARED CHAN BYTE scr!)
  VAL []INT T0 IS [1,0,3,2,5,4,7,6] :
  VAL []INT T1 IS [2,3,0,1,6,7,4,5] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3] :
  [8]CHAN PTCL c0  :
  [8]CHAN PTCL c1  :
  [8]CHAN PTCL c2  :
  PAR i=0 FOR 8
    process(i, c0[i]!, c1[i]!, c2[i]!, c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, scr!)
:
```

18.3 Le routage

Le routage , à partir d'un sommet s vers un sommet destination $d \neq s$ s'appuie sur le calcul du rang du bit de plus faible poids égal à 1 de l'expression $s >< d$. Le calcul de ce rang est fait par la fonction bitfind().
Si le rang est 0 le routage se fait à travers un canal c0(s), s'il vaut 1 à travers un canal c1(s) et vers un canal c2(s) si le rang est 2.
Exemple :
Pour router de 0 vers 7 : $0 >< 7 = 7$ le rang du bit à 1 de plus faible poids vaut 0. On route sur 1 à travers le canal c0(0).
Pour router de 1 vers 7 : $1 >< 7 = 6$ le rang du bit à 1 de plus faible poids vaut 1. On route sur 3 à travers le canal c1(1).
Pour router de 3 vers 7 : $3 >< 7 = 4$ le rang du bit à 1 de plus faible poids vaut 2. On route sur 7 à travers le canal c2(3).
Le tache de routage est partagée par deux processus route.in() qui prend en charge les canaux entrants et route.out() qui dispatche sur les canaux sortants. Si un sommet s est destinataire route.in() l'affiche sinon il communique le flit à route.out via le canal in.out.

```
PROTOCOL STRING IS INT::[]BYTE    :
PROTOCOL PTCL IS INT; INT; STRING :

PROC process(VAL INT i, CHAN PTCL c0.out!,  c1.out!,  c2.out!,
                        c0.in?, c1.in?, c2.in?,SHARED CHAN BYTE screen!)
  PROC route.in(CHAN PTCL out!, c0.i?, c1.i?, c2.i?,
              SHARED CHAN BYTE screen!)
  :

   PROC route.out(VAL INT pid, CHAN PTCL in?, c0.o!, c1.o!, c2.o!)
   :

   CHAN PTCL in.out   :
   PAR
    route.in(    in.out! , c0.in? , c1.in? , c2.in?, screen!)
    route.out(i, in.out? , c0.out!, c1.out!, c2.out!)

:

PROC main(SHARED CHAN BYTE scr!)
  VAL []INT T0 IS [1,0,3,2,5,4,7,6] :
  VAL []INT T1 IS [2,3,0,1,6,7,4,5] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3] :
  [8]CHAN PTCL c0  :
  [8]CHAN PTCL c1  :
  [8]CHAN PTCL c2  :
  PAR i=0 FOR 8
    process(i, c0[i]!, c1[i]!, c2[i]!, c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, scr!)
:
```

Le programme complet :

```
-- ch18_2.occ
--
PROTOCOL STRING IS INT::[]BYTE    :
PROTOCOL PTCL IS INT; INT; STRING :

PROC process(VAL INT i, CHAN PTCL c0.out!,  c1.out!,  c2.out!,
                        c0.in?, c1.in?, c2.in?,SHARED CHAN BYTE screen!)

  PROC route.in(CHAN PTCL out!, c0.i?, c1.i?, c2.i?,
              SHARED CHAN BYTE screen!)

    PROC affiche(VAL INT len, []BYTE buffer, SHARED CHAN BYTE scr!)
      CLAIM scr!
        SEQ
          SEQ i=0 FOR len
            scr! buffer[i]
          scr! '*n'
    :

    VAL INT delai IS 1000000:
    INT emet, recoit, len   :
    INT now0, now1          :
```

```
    TIMER clock              :
    [256]BYTE buffer         :
    INITIAL BOOL encore IS TRUE :
    SEQ
      clock? now0
      --
      WHILE encore
        PRI ALT
          c0.i? emet; recoit; len::buffer
            SEQ
              IF
                i = recoit
                  SEQ
                    affiche(len, buffer, screen!)
                    out! -1; -1; 0::""
                TRUE
                  out! emet; recoit; len::buffer
              encore := FALSE

          c1.i? emet; recoit; len::buffer
            SEQ
              IF
                i = recoit
                  SEQ
                    affiche(len, buffer, screen!)
                    out! -1; -1; 0::""
                TRUE
                  out! emet; recoit; len::buffer
              encore := FALSE

          c2.i? emet; recoit; len::buffer
            SEQ
              IF
                i = recoit
                  SEQ
                    affiche(len, buffer, screen!)
                    out! -1; -1; 0::""
                TRUE
                  out! emet; recoit; len::buffer
              encore := FALSE

          clock? now1
            IF
              (now1-now0) > delai
                SEQ
                  out! -1; -1; 0::""
                  encore := FALSE
              TRUE
                SKIP
  :


  PROC route.out(VAL INT pid, CHAN PTCL in?, c0.o!, c1.o!, c2.o!)
```

```
INT FUNCTION bitfind(VAL INT x, y)
  INT aux , i  :
  BOOL encore  :
  VALOF
    SEQ
      i := 0
      encore := TRUE
      aux := x >< y
      WHILE encore AND (i < 8)
        IF
          (aux BITAND (1 << i)) <> 0
            encore := FALSE
          TRUE
            i := i+1
    RESULT i
:
--
--
INT rang  :
INT emet, recoit, len :
[256]BYTE tampon      :
VAL []BYTE message IS "Route hypercube" :
SEQ
  IF
    pid = 0
      SEQ
        --  le processus 0 emet vers 7
        emet := pid
        recoit := 7
        rang := bitfind(i, recoit)
        CASE rang
          0
            c0.o! emet; recoit; (SIZE message)::message
          1
            c1.o! emet; recoit; (SIZE message)::message
          2
            c2.o! emet; recoit; (SIZE message)::message
    TRUE
      SKIP
  --
  --  reception
  --
  in? emet; recoit; len::tampon
  IF
    len > 0
      SEQ
        rang := bitfind(pid, recoit)
        CASE rang
          0
            c0.o! emet; recoit; len::tampon
          1
            c1.o! emet; recoit; len::tampon
          2
            c2.o! emet; recoit; len::tampon
```

```
        TRUE
          -- la chaine de longueur 0 pour la bonne
          -- terminaison n'est pas traitee.
          SKIP
  :

  CHAN PTCL in.out   :
  PAR
    route.in(    in.out! , c0.in? , c1.in? , c2.in?, screen!)
    route.out(i, in.out? , c0.out!, c1.out!, c2.out!)
:


PROC main(SHARED CHAN BYTE scr!)
  VAL []INT T0 IS [1,0,3,2,5,4,7,6] :
  VAL []INT T1 IS [2,3,0,1,6,7,4,5] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3] :
  [8]CHAN PTCL c0  :
  [8]CHAN PTCL c1  :
  [8]CHAN PTCL c2  :
  PAR i=0 FOR 8
    process(i, c0[i]!, c1[i]!, c2[i]!, c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, scr!)
:
```

18.4 Utiliser le parallélisme des liens

On peut , comme dans la grille torique, améliorer le routage à en routant en parallèle dans des directions différentes à partir d'un sommet s donné. Il faut cependant modifier l'algorithme standart car son utilisation ne conduit pas à des chemins arêtes disjoints et amène à des interblocages.
Par exemple pour router en parallèle de 0 vers 7 dans $H(3)$ avec l'algorithme standart on est amenè à suivre les 3 chemins :

```
c0: 0 -> 1 -> 3 -> 7
c1: 0 -> 2 -> 3 -> 7
c2: 0 -> 4 -> 5 -> 7
```

On remarque que les chemins $c0$ et $c1$ partagent en commun l'arète 3,7.

18.4.1 Routage à distance $k \leq n$ dans $H(n)$

Théorème :
Quels que soient les sommets u et v de $H(n)$ à distance $k \leq n$ il existe k chemins arètes disjoints deux à deux, de longueur k, joignant u et v .

1- Cas particulier où $k = n$:

Si $s\epsilon H(n)$ on posera $B_k(s)$ l'application qui à s fait correspondre le sommet qui ne diffère de s que par le bit de rang k pour $k = 0 \ldots n-1$.
Notons c_0 , c_0 , .. c_{n-1} les n chemins arètes disjoints joignant u et v .

Construction des chemins :
$c_0 : x_0 = u$, $x_1 = B_0(x_0)$, $x_2 = B_1(x_1)$, $v = B_{n-1}(x_{n-1})$
$c_1 : x_0 = u$, $x_1 = B_1(x_0)$, $x_2 = B_2(x_1)$, $v = B_0(x_{n-1})$
...
$c_k : x_0 = u$, $x_1 = B_k(x_0)$, $x_2 = B_{k+1}(x_1)$, $v = B_{k+n-1}(x_{n-1})$
...
$c_{n-1} : x_0 = u$, $x_1 = B_{n-1}(x_0)$, $x_2 = B_0(x_1)$, $v = B_{n-2}(x_{n-1})$

Nota : Les sommes de la forme $k + j$ dans les écritures B_{k+j} sont calculées modulo n.

Exemple dans $H(3)$ avec $s_0 = 101$ et $s_1 = 010$:

c_0 $x_0 = (101)$ $x_1 = (100)$ $x_2 = (110)$ $x_3 = (010)$
c_1 $x_0 = (101)$ $x_1 = (111)$ $x_2 = (011)$ $x_3 = (010)$
c_2 $x_0 = (101)$ $x_1 = (001)$ $x_2 = (000)$ $x_3 = (010)$

L'algorithme de routage à distance n :

```
Soit s.d le sommet destination et s un sommet quelconque :
Si s <> s.d
  debut
       Le flit est lu dans le canal entrant c_{i};
       j = (i+1) modulo n ;
       router le flit dans le canal sortant c_{j};
  fin
```

2- Cas où $k < n$:

Soient u et v deux sommets de $H(n)$ à distance k.
Ils appartiennent à un hypercube de dimension k caractérisé par les sommets de $H(n)$ dont $(n - k)$ bits de leur code binaire sont identiques.
Soient $x_0, \ldots x_{n-k-1}$ les $(n-k)$ bits communs à u et v. Notons $H(n,k)(x_0, \ldots x_{n-k-1})$ l'hypercube de dimension k correspondant à tous les sommets $s \in H(n)$ ayant leur bits de rang $x_0, \ldots x_{n-k-1}$ égaux à ceux de u et v .
$H(n,k)(x_0, \ldots x_{n-k-1})$ est un hypercube isomorphe à $H(k)$.
On applique à $H(n,k)(x_0, \ldots x_{n-k-1})$ l'algorithme de routage sur $H(k)$.
Cet algorithme doit être légèrement modifié pour que les $(n - k)$ bits $x_0, \ldots x_{n-k-1}$ communs à u et v ne soient pas pris en compte.

L'algorithme de routage à distance $k < n$:

```
Soient s.e le sommet emetteur et s.d le sommet destinataire.
Le flit est lu sur le canal c_{k}
Les bits a 1 de la  variable bit.fix := NOT( s.e BITAND s.d) donnent
le rang des bits communs a  s.e et s.d.
On saute ces bits communs en executant la boucle :
i := 1
WHILE ((1 << ((k+i)REM n)) BITAND bit.fix ) <> 0
  i := i+1
router sur le canal c_{k+i}
```

Le programme de test dans $H(4)$. Le sommet 5 route vers 8 :

```
-- ch18_3.occ
--
PROTOCOL STRING IS INT::[]BYTE      :
PROTOCOL PTCL   IS INT; INT; STRING :


PROC process(VAL INT pid,em,rec, CHAN PTCL c0.out!, c1.out!, c2.out!, c3.out!,
                        c0.in?, c1.in?, c2.in?, c3.in?, SHARED CHAN BYTE screen!)

  PROC affiche(VAL INT pi, len, []BYTE buffer, SHARED CHAN BYTE scr!)
    CLAIM scr!
      SEQ
        IF
          pi < 10
            scr! '0'+ (BYTE pi)
          TRUE
            scr! 'A' + ((BYTE pi)-10)

        scr! ' '
        SEQ i=0 FOR len
          scr! buffer[i]
        scr! '*n'
  :


  VAL INT delai IS 1000000    :
  INT now0, now1              :
  TIMER clock                 :
  [256]BYTE buffer            :
  INITIAL BOOL encore IS TRUE :
  INITIAL INT nb.message IS 0 :
  INT emet, recoit, len       :
  VAL [4][]BYTE message IS ["message type 0",
                            "message type 1",
                            "message type 2",
                            "message type 3"]:

  [4]CHAN PTCL c.out IS [c0.out,c1.out,c2.out,c3.out]   :
  [4]CHAN PTCL c.in  IS [c0.in, c1.in, c2.in, c3.in]    :
  IF
    pid = em
      --  *************** emission *************
      SEQ
        emet   := em
        recoit := rec
        --
        PAR i= 0 FOR 4
          INT bit.fix :
          SEQ
            bit.fix := BITNOT (emet >< recoit)
            IF
              ((1<< i) BITAND bit.fix ) <> 0
                SKIP
              TRUE
```

```
              c.out[i]! emet; recoit ; (SIZE message[i])::message[i]

    pid <> em
      SEQ
        --  *************** reception ************
        clock? now0
        WHILE encore
          PRI ALT
            ALT k=0 FOR 4
              c.in[k]? emet; recoit; len::buffer
                SEQ
                  affiche(pid,len, buffer, screen!)
                  IF
                    pid = recoit
                      SEQ
                        nb.message := nb.message + 1
                        IF
                          nb.message = 4
                            encore := FALSE
                          TRUE
                            SKIP
                    TRUE
                      INT bit.fix         :
                      INITIAL INT i IS 1  :
                      SEQ
                        bit.fix := BITNOT (emet >< recoit)
                        WHILE ((1 << ((k+i)REM 4)) BITAND bit.fix ) <> 0
                          i := i+1
                        c.out[(k+i) REM 4]! emet; recoit; len::buffer
                        encore := FALSE

            clock? now1
              IF
                (now1-now0) > delai
                  encore := FALSE
                TRUE
                  SKIP
:


PROC main(CHAN BYTE kbd, SHARED CHAN BYTE scr!)
  VAL []INT T1 IS [2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13] :
  VAL []INT T0 IS [1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11] :
  VAL []INT T3 IS [8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7] :
  [16]CHAN PTCL c0   :
  [16]CHAN PTCL c1   :
  [16]CHAN PTCL c2   :
  [16]CHAN PTCL c3   :
  PAR i=0 FOR 16
    process(i,5,8, c0[i]!, c1[i]!, c2[i]!,c3[i]!,
                   c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, c3[T3[i]]?, scr!)
:
```

18.4.2 Cas général

Théorème :
Quels que soient les sommets u et v de $H(n)$ il existe n chemins arêtes disjoints deux à deux joignant u et v.
Si u et v sont à distance k ils sont joints par k chemins de longueur k et par $(n-k)$ chemins de longueur $(k+2)$.

Soient u et v deux sommets de $H(n)$ à distance k.
Comme il a été vu précédemment ils appartiennent à l'hypercube $H(n,k)(x_0, \ldots x_{n-k-1})$ de dimension k.

Soient $x_0, \ldots x_{n-k-1}$ les $(n-k)$ bits communs à u et v.
Notons $I_{n,k} = x_0, \ldots, x_{n-k-1}$.
Deux cas de figure se présentent :

Premier cas : le routage se fait dans $H(n,k)(x_0, \ldots x_{n-k-1})$.
En routant dans $H(n,k)(x_0, \ldots x_{n-k-1})$ on obtient k chemins de longueur k. Pour ce faire il suffit , à partir de u de router sur toutes les directions qui ne sont pas dans $I_{n,k}$. D'après le resultat précédant nous savons qu'il existe k chemins arêtes disjoints dans $H(n,k)$ joignant deux sommets distants de k.

Deuxième cas : Le routage se fait dans des directions associées à $I_{n,k}$.
Soit $i \in I_{n,k}$. Notons u_i et v_i les homologues de u et v dans $H(n)$ qui ne diffèrent de ces derniers que par le bit de rang i.
On remarque que u_i et v_i sont également à la distance k.
Le routage se fait en trois étapes :
1 On route de u vers u_i
2 On route de u_i vers v_i dans un hypercube de dimension k .
3 On route de v_i vers v.
Ce routage se fait en suivant un chemin de longueur : $1 + k + 1$ soit : $(k + 2)$.

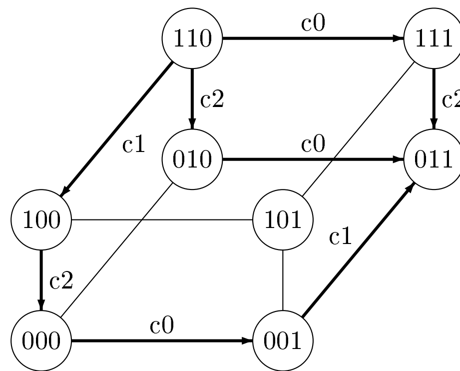


FIGURE 18.2 – Routage dans $H(3)$ à distance de 2

Le programme qui suit montre l'implémentation des résultats qui viennent d'être exposés de routage de deux sommets quelconques à distance k dans $H(n)$.
On notera que le protocole PTCL comporte un champ supplémentaire référençé **tag**.

Ce champ facilite le routage en informant un sommet s sur la nature du flit qui transite par ce sommet.
Si le flit est relatif à un trajet à distance k dans un hypercube $H(n,k)(x_0,\dots x_{n-k-1})$ il vaut -1 sinon il code le rang du bit dans $I_{n,k}$ associé à un chemin de longueur $(k+2)$.
Le codage de ce champ est fait par le processus émetteur pour les n directions possibles.

```
-- ch18_4.occ
--
#USE "course.lib"

PROTOCOL STRING IS INT::[]BYTE           :
PROTOCOL PTCL   IS INT; INT; INT; STRING :


INT FUNCTION chg.bit(VAL INT x, rang)
  INT aux, result  :
  VALOF
    SEQ
      aux := 1 << rang
      IF
        (x BITAND aux) = 0
          result := x BITOR aux
        TRUE
          result := x BITAND (BITNOT aux)
    RESULT result
:


PROC process(VAL INT pid,em,rec, CHAN PTCL c0.out!, c1.out!, c2.out!, c3.out!,
                        c0.in?, c1.in?, c2.in?, c3.in?, SHARED CHAN BYTE screen!)

  PROC affiche(VAL INT pi, len, []BYTE buffer, SHARED CHAN BYTE scr!)
    CLAIM scr!
      SEQ
        IF
          pi < 10
            scr! '0'+ (BYTE pi)
          TRUE
            scr! 'A' + ((BYTE pi)-10)

        scr! ' '
        SEQ i=0 FOR len
          scr! buffer[i]
        scr! '*n'
  :


  VAL INT delai IS 500000    :
  INT now0, now1              :
  TIMER clock                 :
  [256]BYTE buffer            :
  INT tag                     :
  [4]INT emet, recoit         :
```

```
INT r.emet, r.recoit        :
INT len, bit.fix            :
[4]INT e.tag                :
VAL [4][]BYTE message IS ["message type 0",
                          "message type 1",
                          "message type 2",
                          "message type 3"]:

[4]CHAN PTCL c.out IS [c0.out,c1.out,c2.out,c3.out]   :
[4]CHAN PTCL c.in  IS [c0.in, c1.in, c2.in, c3.in]    :
SEQ
  --  ******************** emission **********************
  IF
    pid = em
      SEQ
        SEQ i=0 FOR 4
          SEQ
            bit.fix := BITNOT (em >< rec)
            IF
              ((1<< i) BITAND bit.fix ) <> 0
                SEQ
                  emet[i]   := chg.bit(em,i)
                  recoit[i] := chg.bit(rec,i)
                  e.tag[i]  := i
              TRUE
                SEQ
                  emet[i]   := em
                  recoit[i] := rec
                  e.tag[i]  := -1
        PAR i= 0 FOR 4
          c.out[i]! e.tag[i]; emet[i]; recoit[i] ; (SIZE message[i])::message[i]
    --
    --  ************************ reception **********************

    pid <> em
      SEQ
        clock? now0
        INITIAL BOOL encore IS TRUE :
        INITIAL INT nb.message IS 0 :
        WHILE encore
          PRI ALT
            ALT k=0 FOR 4
              c.in[k]? tag; r.emet; r.recoit; len::buffer
                SEQ
                  affiche(pid,len, buffer, screen!)
                  --
                  --
                  IF
                    (pid = r.recoit) AND (tag < 0)
                      SEQ
                        nb.message := nb.message + 1
                        IF
                          nb.message = 4
                            encore := FALSE
```

```
                          TRUE
                            SKIP
                    (pid = r.recoit) AND (tag >= 0)
                      SEQ
                        r.recoit := chg.bit(r.recoit,tag)
                        c.out[tag]! -1; r.emet; r.recoit; len::buffer
                        encore := FALSE

                    pid <> r.recoit
                      INITIAL INT i IS 1  :
                      SEQ
                        bit.fix := BITNOT (r.emet >< r.recoit)
                        WHILE ((1 << ((k+i)REM 4)) BITAND bit.fix ) <> 0
                          i := i+1
                        c.out[(k+i) REM 4]! tag; r.emet; r.recoit; len::buffer
                        encore := FALSE

              clock? now1
                IF
                  (now1-now0) > delai
                    encore := FALSE
                  TRUE
                    SKIP
:


PROC main(CHAN BYTE kbd?, SHARED CHAN BYTE scr!)
  VAL []INT T1 IS [2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13] :
  VAL []INT T0 IS [1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11] :
  VAL []INT T3 IS [8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7] :
  [16]CHAN PTCL c0   :
  [16]CHAN PTCL c1   :
  [16]CHAN PTCL c2   :
  [16]CHAN PTCL c3   :
  INT emet, recoit   :
  BOOL ok            :
  SEQ
    CLAIM scr!
      SEQ
        out.string("emetteur: ",0, scr!)
        flush(scr!)
        in.int(emet,2, kbd?, scr!)
    CLAIM scr!
      SEQ
        flush(scr!)
        out.string("*nrecepteur: ",0, scr!)
        flush(scr!)
        in.int(recoit,2,kbd?, scr!)
        flush(scr!)
        out.string("*n*n",0, scr!)
    --
    PAR i=0 FOR 16
      process(i,emet,recoit, c0[i]!, c1[i]!, c2[i]!, c3[i]!,
```

```
                c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, c3[T3[i]]?, scr!)
:
```

Exemple d'exécution

```
emetteur: 6
recepteur: 3

7 message type 0
E message type 3
2 message type 2
4 message type 1
3 message type 0
F message type 3
0 message type 1
3 message type 2
B message type 3
1 message type 1
3 message type 3
3 message type 1
```

Dans cet exemple l'emetteur est le sommet $6 = 0110$ et le recepteur le sommet $3 = 0011$. L'ensemble des bits communs à 6 et 3 est $I = 1, 3$.
On obtient 2 chemins de longueur 2 lorsque l'on route dans les directions $0, 2$. Ces chemins sont marqués $type0$ et $type2$ et l'on obtient 2 chemins de longueur 4 lorsque l'on route dans les directions $1, 3$ de I. Ces chemins sont marqué $type1$ et $type3$. L'intitulé "7 message type 0" signifie que le sommet 7 a reçu un message de type 0.

18.5 Diffusion

Une diffusion est l'émission par un processus dans un réseau, d'un message en direction de tous les autres processus sur le réseau.
Un arbre de recouvrement d'un graphe est un arbre dont les sommets coïncident avec ceux de ce graphe. Les arbres de recouvrement sont habituellement associés à des diffusions.
Plusieurs types d'arbres de recouvrement sont associés à l'hypercube. Dans ce qui suit nous nous intéressons aux arbres binaires de recouvrement de l'hypercube.
La définition de l'arbre de recouvrement $A(n)$ de $H(n)$ est récursive .
Définition de $A(n)$
$n = 1$:
On joint un sommet s à son homologue dans $H(n)$. L'arbre $A(1)$ de sommet s possède 1 arète.
$n > 1$
Etant donné un arbre $A(n-1)$ de sommet s , on le duplique. Les deux arbres donnent naissance à l'arbre $A(n)$ lorsque l'on joint s à son homologue dans $H(n)$.
La figure ci jointe illustre la construction de $A(3)$ de sommet (000).

Algorithme de routage :
En examinant la structure de l'arbre binaire de recouvrement on en déduit l'algorithme de routage :
Soit un sommet $s = (s_{n-1}, \ldots, s_0)$ atteint par un message émis par le sommet $u = (u_{n-1}, \ldots, u_0)$.

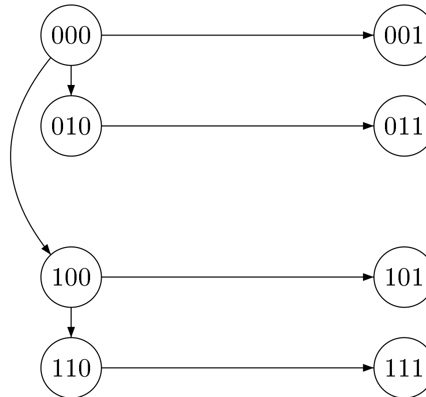


FIGURE 18.3 – Un arbre binaire de recouvrement de $H(3)$

$i \leftarrow 0$
tant que $s_i = u_i$
debut
router dans la direction i ;
$i \leftarrow i + 1$;
fin

Le programme de test :

```
-- ch18_5.occ
--
PROTOCOL STRING IS INT::[]BYTE    :
PROTOCOL PTCL IS INT; STRING :

PROC process(VAL INT pid, CHAN PTCL c0.o!,  c1.o!,  c2.o!,
                        c0.in?, c1.in?, c2.in?,SHARED CHAN BYTE screen!)


  VAL INT delai IS 1000000    :
  VAL []BYTE message IS " Diffuse sur  hypercube vers : " :
  INITIAL BOOL encore IS TRUE :
  INT len, emet               :
  [256]BYTE buffer            :
  --  les canaux sont renommes
  --  pour beneficier de l'indexation
  [3]CHAN PTCL c.out IS [c0.o,c1.o,c2.o]      :
  [3]CHAN PTCL c.in  IS [c0.in,c1.in,c2.in]   :
  --  --  emission
  SEQ
    IF
      pid = 3
        SEQ
```

```
          --  le sommet 3 lance la diffusion puis se termine
          emet := pid
          PAR i=0 FOR 3
            c.out[i]! emet; (SIZE message)::message
          encore := FALSE

      TRUE
        SKIP
    --
    --  reception puis diffusion par un sommet different de 3
    --
    WHILE encore
      ALT k=0 FOR 3
        c.in[k]? emet; len::buffer
          SEQ
            CLAIM screen!
              SEQ
                screen! (BYTE emet) + '0'
                SEQ j=0 FOR len
                  screen! buffer[j]
                screen! (BYTE pid) + '0'
                screen! '*n'

            INT aux1, aux2                :
            INITIAL BOOL encore1 IS TRUE  :
            INITIAL INT rang IS 0         :
            WHILE encore1 AND (rang < 3)
              SEQ
                aux1 := pid  BITAND (1 << rang)
                aux2 := emet BITAND (1 << rang)
                IF
                  aux1 = aux2
                    SEQ
                      c.out[rang]! emet; (SIZE message)::message
                      rang := rang + 1
                  TRUE
                    encore1 := FALSE
            encore := FALSE
:


PROC main(SHARED CHAN BYTE scr!)
  VAL []INT T0 IS [1,0,3,2,5,4,7,6] :
  VAL []INT T1 IS [2,3,0,1,6,7,4,5] :
  VAL []INT T2 IS [4,5,6,7,0,1,2,3] :
  [8]CHAN PTCL c0  :
  [8]CHAN PTCL c1  :
  [8]CHAN PTCL c2  :
  PAR i=0 FOR 8
    process(i, c0[i]!, c1[i]!, c2[i]!, c0[T0[i]]?, c1[T1[i]]?, c2[T2[i]]?, scr!)
:
```

Chapitre 19

Le graphe complet

Dans un réseau organisé en graphe complet chaque processus est relié à tous les autres. Un tel réseau, s'il a n processus, comporte alors $A(n) = n(n-1)/2$ canaux.
On se propose dans ce qui suit de créer dynamiquement un réseau graphe complet constitué de n processus ou n est un paramètre arbitraire. Dans un tel réseau les canaux sont des canaux mobiles crées à l'exécution.
On montre en outre comment implémanter un échange total pour un réseau de ce type .

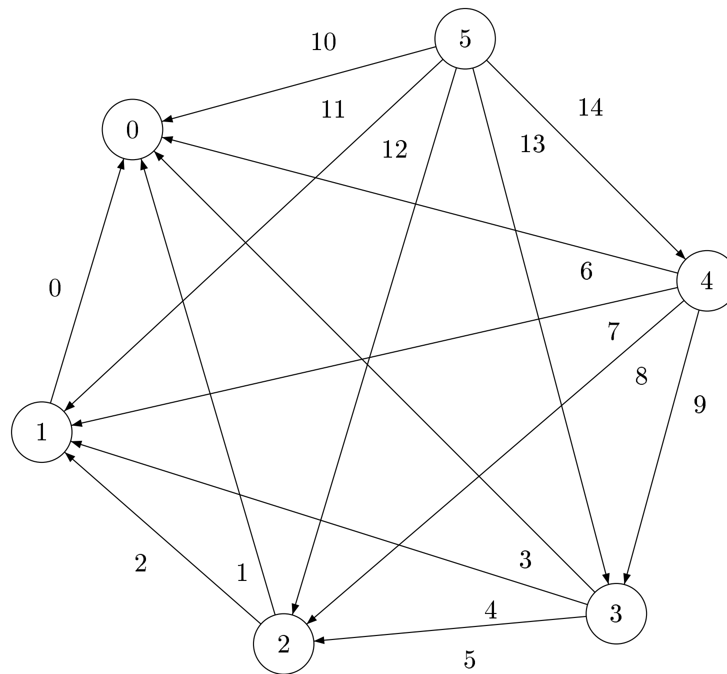


FIGURE 19.1 – Un graphe complet d'ordre 6

19.1 Le codage des canaux mobiles

On notera $C(n)$ le réseau graphe complet orienté à n processus .
Un canal mobile joignant le processus i au processus j où i est le bout serveur et j le bout client définit l'arète orientée (i, j). (La figure 19.1 illustre $C(6)$).

Le codage des canaux est défini récursivement comme suit :
$C(1)$ ayant un sommet , ce dernier est indexé 0 et possède 0 canaux.
$C(n)$ possède n sommets indéxés de 0 à $(n-1)$ et $A(n) = n(n-1)/2$ canaux numérotés de 0 à $A(n) - 1$.

$C(n+1)$ se construit à partir de $C(n)$ en ajoutant un sommet indexé n et :
le canal joignant le nouveau sommet n au sommet 0 est numéroté $A(n) + 0$.
0 est client et n est serveur
le canal joignant le nouveau sommet n au sommet 1 est numéroté $A(n) + 1$.
1 est client et n est serveur
—
le canal joignant le nouveau sommet n au sommet k est numéroté $A(n) + k$.
k est client et n est serveur

Dans $C(n)$ le processus indexé i possède i bouts serveurs et $(n-1) - i$ bouts clients. En particulier le processus 0 ne possède que des bouts clients et le processus $(n-1)$ ne possède que des bouts serveurs.

La structure du réseau $C(n)$ est codée par la matrice chan.link[n][n].
Cette dernière est construite lors de la phase d'initialisation (CF ci dessous).
Pour i fixé et $i > j$ chanel.link[i][j] donne les numéros des bouts serveurs aboutissant au processus i.
Pour i fixé et $i < j$ chanel.link[i][j] donne les numéros des bouts clients aboutissant au processus i.
La valeur -1 attribuée aux éléments diagonaux de chanel.link est arbitraire.

$$\begin{bmatrix} -1 & 0 & 1 & 3 & 6 & 10 \\ 0 & -1 & 2 & 4 & 7 & 11 \\ 1 & 2 & -1 & 5 & 8 & 12 \\ 3 & 4 & 5 & -1 & 9 & 13 \\ 6 & 7 & 8 & 9 & -1 & 14 \\ 10 & 11 & 12 & 13 & 14 & -1 \end{bmatrix}$$

La matrice chan.link assoçiée à $C(6)$

19.2 Le processus init.net

Le nombre, nb.proc, de processus à interconnecter est passé en paramètre au processus init.net() dont le rôle est de créer les $A(n)$ bouts serveurs et les $A(n)$ bouts clients puis de les distribuer aux processus lorsque ces derniers en font la requète.

Les paramètres passés à init.net() :

nb.proc : nombre de processus.
request : Un processus qui veut se connecter écrit son index dans ce canal partagé.
cli ! : init.net() transmet les bouts clients à un processus qui a émis une requète de connection dans ce canal paratagé.
serv ! : init.net() transmet les bouts serveurs à un processus qui a émis une requète de connection dans ce canal partagé.

On distingue deux phases dans l'activite de init.net().

19.2.1 Phase un de init.net()

Dans cette phase le processus crée les bouts des canaux mobiles puis la matrice chan.link.
Les bouts de type serveurs sont stockés dans le tableau s.link[].
Les bouts de type clients sont stockés dans le tableau c.link[].

19.2.2 Phase deux de init.net()

Dans cette phase init.net() attend les nb.proc requètes issues des processus.
Un processus qui émet une requète écrit son index proc.id qui est lu par init.net() dans le canal partagé request.
init.net() écrit alors les proc.id bouts serveurs indexés correctement grace à la matrice chan.link au processus demandeur à travers le canal partagé serv.
De même les (nb.proc-(1+proc.id)) bouts clients, correctements indexés, sont transmis à travers le canal partagé cli au processus demandeur.

Exemple issus du réseau $C(6)$.

Le processus demandeur est le processus 3.
Il reçoit les bouts serveurs d'indices 3,4, 5 sur le canal cli.
Il reçoit les bouts clients d'indices 9, 13 sur le canal serv.

```
-- ch19_1.occ
--
 PROC init.net(VAL INT nb.proc, CHAN INT request?, CHAN LINK! cli!, CHAN LINK? serv!)
  INT nb.link  :
  INT proc.id  :

  MOBILE []LINK? s.link       :
  MOBILE []LINK! c.link       :
  MOBILE [][]INT chan.link    :

  --  Phase Un ****************************************
  --
  SEQ
    nb.link := (nb.proc * (nb.proc - 1)) / 2

    s.link := MOBILE [nb.link]LINK?
    c.link := MOBILE [nb.link]LINK!
```

```
    SEQ i=0 FOR nb.link
      s.link[i], c.link[i] := MOBILE LINK

    chan.link := MOBILE [nb.proc][nb.proc]INT
    SEQ i=0 FOR nb.proc
      chan.link[i][i]:= -1

    INITIAL INT m IS 0  :
    SEQ i=0 FOR nb.proc
      SEQ j=0 FOR i
        SEQ
          chan.link[i][j]:= m
          chan.link[j][i]:= m
          m := m+1

    --
    --Phase Deux **************************************

    INITIAL INT nb.request IS 0 :
    WHILE nb.request < nb.proc
      SEQ
        request? proc.id
        SEQ
          SEQ i=0 FOR proc.id
            serv! s.link[chan.link[proc.id][i]]
          SEQ i=proc.id FOR (nb.proc-(1+proc.id))
            cli! c.link[chan.link[proc.id][i+1]]
          nb.request := nb.request + 1

:
```

19.3 Les processus process()

Ces processus sont interconnectés par le réseau graphe complet $C(n)$.

19.3.1 La matrice c.chan.link

On vient de voir que suite à sa requète un processus i reçoit du processus init.net() les i bouts serveurs et les $(n-1)-i$ bouts clients qui le relie aux autres processus du réseau.
A leur réception les i bouts serveurs sont stockés dans le tableau s.link[i] dans l'ordre de leur réception.
Les bouts clients sont stockés dans le tableau c.link[$(n-1)-i$] également dans l'ordre de leur réception.

Par exemple le processus 4 stocke dans s.link[0] le bout serveur d'indice 6, dans s.link[1] le bout serveur d'indice 7, dans s.link[2] le bout serveur d'indice 8 et dans s.link[3] le bout serveur d'indice 9.
Ce même processus stocke dans c.link[0] le bout client d'indice 14.

La matrice c.chan.link permet de trouver le bon canal sur lequel un processus i doit lire

s'il connait l'indice k du processus qui lui écrit. Son écriture facilite la compréhension des échanges entre processus.
Soit n la valeur de c.chan.link[i][k] .
Si $i < k$ alors le processus i lit sur le canal c.link[n].
Si $i > k$ alors le processus i lit sur le canal s.link[n].
Par exemple si le processus 4 sait que le processus 3 lui écrit alors n vaut 3 et le processus i lit le canal s.link[3] qui correspond au canal d'indice 9.
On voit donc que c.chan.link[i][k] $= k$ pour $k < i$ et que c.chan.link[i][k] $= (k-i)-1$ pour $k > i$.

$$\begin{bmatrix} -1 & 0 & 1 & 2 & 3 & 4 \\ 0 & -1 & 0 & 1 & 2 & 3 \\ 0 & 1 & -1 & 0 & 1 & 2 \\ 0 & 1 & 2 & -1 & 0 & 1 \\ 0 & 1 & 2 & 3 & -1 & 0 \\ 0 & 1 & 2 & 3 & 4 & -1 \end{bmatrix}$$

La matrice c.chan.link correspondant à $C(6)$

19.3.2 Phase un de process()

Les paramètres passés à process() :

id : index affecté au processus process().
nb.process : nombre de processus interconnectés.
request ! : canal partagé au moyen duquel process() effectue sa requête des bouts de canaux mobiles auprès de init.net().
c ? : canal partagé par lequel les bouts de type client sont envoyés à process() par init.net()
s ? : canal partagé par lequel les bouts de type serveur sont envoyés à process() par init.net()
out ! : canal partagé servant a l'affichage sur l'écran.
bar : barrière de synchronisation associée à l'établissement complet du réseau.
Dans cette phase le processus process() initialise la matrice c.chan.link puis émet une requête a init.net() sur le canal request en écrivant son index.
Les tableaux s.link[] et c.link[] destinés a recevoir les bouts des canaux mobiles sont constitués.
Finalement en se mettant en lecture sur le canal s le processus lit les bouts serveurs émis par init.link() et sur le canal c il lit les bouts clients.
A l'issue de cette phase le processus process() est relié aux autres membres du réseau dès lors qu'il a franchi la barrière de synchronisation bar.

19.3.3 L'échange total

Il y a échange total dans un réseau dès que chaque processus envoie un message à tous les autres membres du réseau. Dans l'exemple qui suit chaque processus écrit son index en direction de tous les autres.

L'écriture en direction des autres membres du réseau se fait en écrivant sur le canal out des bouts de type serveur et en écrivant sur le canal in des bouts de type client.

```
  PAR
    PAR i=0 FOR lnk.s
      s.link[i][out]! my.coord
    PAR i=0 FOR lnk.c
      c.link[i][in]!  my.coord
```

La lecture d'un message émis par un processus d'index k utilise la matrice c.chan.link.

```
 INT n           :
 INT coord       :
 SEQ
   n := c.chan.link[id][k]
   IF
     id < k
       SEQ
          c.link[n][out]? coord

     id > k
        SEQ
          s.link[n][in]? coord
```

Le texte complet du processus process() :

```
-- ch19_2.occ
--
PROC process(VAL INT id, VAL INT nb.process,  SHARED CHAN INT request!,
              SHARED CHAN LINK! c?, SHARED CHAN LINK? s?,
              SHARED CHAN BYTE out!, BARRIER bar)
  MOBILE []LINK? s.link             :
  MOBILE []LINK! c.link             :
  MOBILE [][]INT c.chan.link        :
  INT my.coord                     :
  INT lnk.c, lnk.s                  :
  INT emet                          :
  SEQ
    my.coord := id
    lnk.s := id
    lnk.c := nb.process -(1+id)
    emet := 2
    --  Phase Un
    --
    --  initialise la matrice c.chan.link

    c.chan.link := MOBILE [nb.process][nb.process]INT
    SEQ i=0 FOR nb.process
      c.chan.link[i][i]:= -1

    INT m :
    SEQ i=0 FOR  nb.process
      SEQ
```

```
      m := 0
      SEQ j=0 FOR i
        SEQ
          c.chan.link[i][j] := m
          m := m+1
      m:= 0
      SEQ j= (i+1) FOR (nb.process -(i+1))
        SEQ
          c.chan.link[i][j] := m
          m := m+1

--
-- demande des bouts des canaux mobiles aupres
-- de init.net()


  CLAIM request!
    SEQ
      request! id
      SEQ
        s.link := MOBILE [lnk.s]LINK?
        c.link := MOBILE [lnk.c]LINK!
        CLAIM s?
          SEQ i=0 FOR lnk.s
            s? s.link[i]
        CLAIM c?
          SEQ j=0 FOR lnk.c
            c? c.link[j]

  SYNC bar

  --  **************************************
  --  Procede a un echange total
  --  **************************************

  PAR k=0 FOR nb.process
    IF
      id = k
      --  l'emetteur est id
        PAR
          PAR i=0 FOR lnk.s
            s.link[i][out]! my.coord
          PAR i=0 FOR lnk.c
            c.link[i][in]!  my.coord

      id <> k
      --   est emis par  k
        INT n           :
        INT coord       :
        SEQ
          n := c.chan.link[id][k]
          IF
            id < k
              SEQ
```

```
                  c.link[n][out]? coord

              id > k
                SEQ
                  s.link[n][in]? coord
            CLAIM out!
              SEQ
                out! '0' + (BYTE my.coord)
                out! ':'
                out! '0' + (BYTE coord)
                out! '*n'
:
```

Le texte du main() qui crée les canaux et lance les processus :

```
PROC main(SHARED  CHAN BYTE scr!)
  INT nb.proc                   :
  BARRIER bar                   :
  SHARED! CHAN INT request      :
  SHARED? CHAN LINK! cli        :
  SHARED? CHAN LINK? serv       :
  SEQ
    nb.proc := 4
    PAR
      init.net(nb.proc, request?, cli!, serv! )
      PAR i=0 FOR nb.proc BARRIER bar
        process(i, nb.proc,request!, cli?, serv?, scr!, bar)
:
```

Chapitre 20

Arithmétique

20.1 L'algorithme d'addition à retenue anticipée

L'algorithme usuel de calcul de la somme de deux entiers ne se prête pas à la parallélisation. En effet toute étape de son calcul est tributaire de la valeur de la retenue qui n'est disponible qu'à l'issue de l'étape précédente.
L'algorithme d'addition à retenue anticipée est basé sur l'utilisation d'un arbre binaire de profondeur n et ayant donc 2^n feuilles . L'addition se fait en $2 * n + 1$ étapes et porte sur des entiers codés sur 2^n bits. Les bits de même rang de chaque opérande sont traités par une feuille de l'arbre.
Pour chaque sommet dans l'arbre on notera fils.gauche et fils.droit ses successeurs (s'il en a) et pere le sommet (s'il existe) dont il est un des fils (droit ou gauche).

Dans l'algorithme on distinguera le rôle joué respectivement par les processus situés aux feuilles de l'arbre, les processus associés à des sommets internes et finalement le processus racine.
Pour chacun des processus l'algorithme se décompose en deux phases : une phase de montée et une phase de descente.

1- **processus feuilles** :

La phase de montée :
Au niveau des feuilles la phase de montée consiste à calculer un code comportant 3 valeurs : s , p et g calculées en fonction des valeurs des bits de même rang de chacun des deux opérandes.
Ces codes sont déterminés par : si les deux bits valents 0 le code est s (pour stop) , si les deux bits valent 1 le code est g (pour génère) et p (pour propage) dans les autres cas.
Considérons les entiers 5 = (0101) et 3 = (0011) codés sur 4 bits.
Les bits de rang k de chaque opérande sont affectés à la feuille de même rang.
La somme s modulo deux sur ces deux bits est calculée et un code : le code spg est déterminé par :

```
spg (1,1) = g              generation d'un report
spg(1,0) = spg(0,1) = p    propagation d'un report
spg(0,0) = s               stoppe la propagation d'un report
```

Voir la figure : 20.1 Etape 1.

```
rangs des bits     3 2 1 0

operande 1         0 1 0 1
operande 2         0 0 1 1

somme modulo 2     0 1 1 0

codes              s p p g
```

Chaque processus feuille écrit le code spg au processus pere .
Voir la figure : 20.1 Etape 2

La phase de descente :

Chaque processus feuille lit les codes émis par le processus pere. Ces codes remplacent l'actuel code si ce dernier est **p** sinon l'actuel code est inchangé.
Voir la figure : 20.2 Etape 3

Cette lecture s'arrète lorsqu'un code de fin est reçu. Le calcul final :

Au terme de la phase de descente le code d'un processus feuille est soit **s** soit **g** qui est converti en les valeurs 0 (resp 1).
Ce code est additionné modulo deux à la valeur s calculée en phase de montée pour donner le bit de rang k de la somme puis le processus se termine.

2- **processus internes** :

La phase de montée :

1 .Le processus lit les deux code spg émis par fils.gauche et fils.droit.
Voir la figure : 20.1 Etape 2
2. le processus écrit le code svg émis par fils.droit à fils.gauche et écrit le code **p** à fils.droit.Cette opération est décisive pour propager vers la gauche les reports éventuels.
Voir la figure : 20.2 Etape 3
3. Le processus calcule $w = u \otimes v$ où u est le code svg écrit par fils.gauche et où v est le code svg écrit par fils.droit.
Voir la figure : 20.2 Etape 4

L'opérateur $\otimes$ est défini par la table :.

$\otimes$	s	p	g
s	s	s	s
p	s	p	g
g	g	g	g

Le sens de l'opérateur $\otimes$ est simple :
(s,x) = s car si un report est propagé par x il est stoppé par s et ce $\forall x$.
(p,g) = g car le report de g sera propagé par p.
(g,x) = g car un report est généré par g. etc ...

On notera que $\otimes$ est associatif mais n'est pas commutatif.

4. Le processus écrit au processus pere la valeur de w.
Voir la figure : 20.3 Etape 5

La phase de descente :

5. Le processus lit les code émis par le processus pere.
Voir la figure : 20.3 Etape 6 ; 20.5 Etape 9

6. Le processus répercute sur fils.gauche et fils.droit les codes lus à l'étape précédente (5). Voir la figure : 20.4 Etape 7 ; 20.5 Etape 10

7. Le processus se termine lorsqu'il recoit un code de fin.

le processus racine

La phase de montée :

1. Le processus lit les codes svg u , v émis par fils.gauche (resp fils.droit).
Voir la figure : 20.3 Etape 5

2. le processus écrit le code svg v émis par fils.droit à fils.gauche et écrit le code **p** à fils.droit.
Voir la figure : 20.3 Etape 6

3. Le processus calcule $w = u \otimes v$ où u est le code svg écrit par fils.gauche et où v est le code svg écrit par fils.droit.
Voir la figure : 20.4 Etape 8

4. Le processus écrit w à fils.gauche et à fils.droit.
Voir la figure : 20.5 Etape 9

5. Le processus écrit le code de fin à fils.gauche et à fils.droit et se termine.

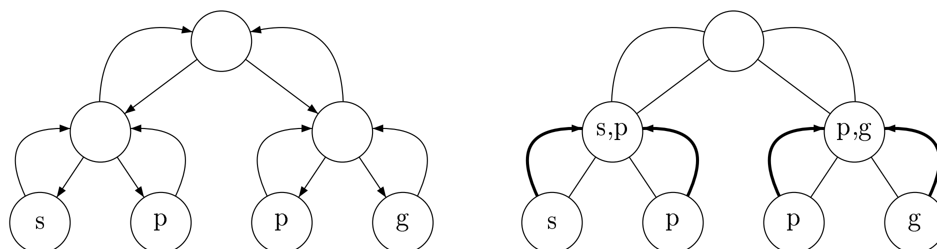


FIGURE 20.1 – Etapes 1 et 2

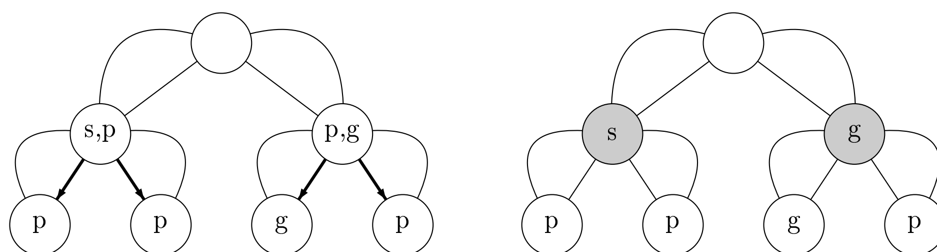


FIGURE 20.2 – Etapes 3 et 4

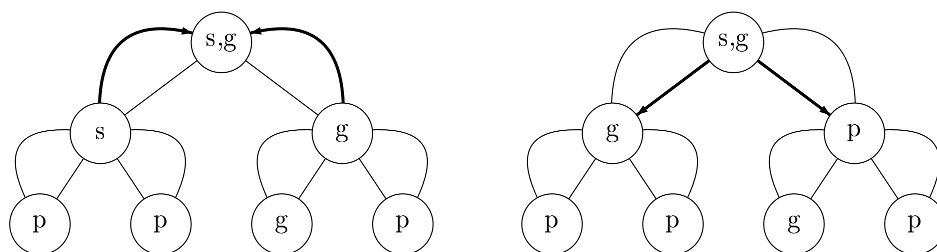


FIGURE 20.3 – Les etapes 5 et 6

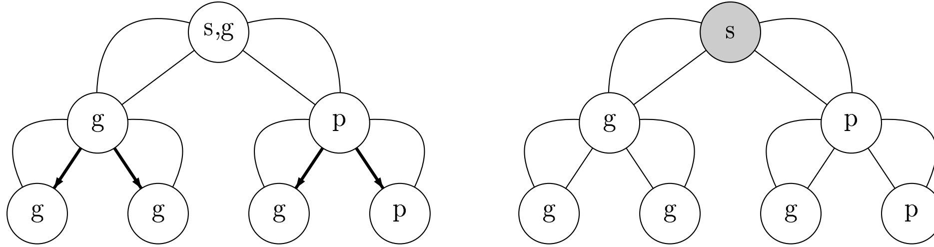


FIGURE 20.4 – Les etapes 7 et 8

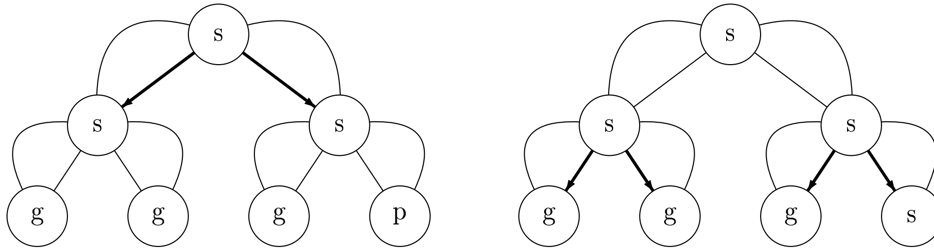


FIGURE 20.5 – Les etapes 9 et 10

20.2 Mise en oeuvre de l'algorithme

20.2.1 le codage des canaux

Le programme traite de l'addition de deux entiers codés sur 16 bits.
L'algorithme d'addition à retenue anticipée nécessite un arbre binaire de profondeur 4. Cet arbre posséde 31 sommets et 30 arêtes.
Les sommets sont numérotés de 0 à 31 à compter de la racine puis en numérotant de la gauche vers la droite en descendant .Ce qui donne :
$[0]$; $[1, 2]$; $[3, 4, 5, 6]$; $[7, 8, 9, 10, 11, 12, 13, 14]$; $[15, ..31]$.
Les informations circulant des feuilles vers la racine et de la racine vers les feuilles les arêtes sont donc bidirectionnelles.
Ces arêtes sont notées $fp[i]$ lorsqu'elles joignent un sommet i à son pere . Ces arêtes sont notées $pf[i]$ lorsqu'elles joignent un pere à un de ses successeurs i (droit ou gauche). Bien que le nombre d'arêtes soit de 30, des tableaux de canaux de dimension 31 sont utilisés pour simplifier l'écriture.
Avec ces conventions les indices des canaux joignant un sommet i à ses successeurs droit et gauche valent $pf[2 * i + 1]$ et $pf[2 * i + 2]$.
Un processus associé à un sommet interne i est relié à son pere et ses deux fils par 6 canaux :

fp[(2*i)+1] ?, fp[(2*i)+2] ?, pf[(2*i)+1] !, pf[(2*i)+2] !, pf[i] ?, fp[i] !
qui correspondent aux lectures de ses deux fils, aux écritures vers ses deux fils, à la lecture du pere et à l'écriture vers le pere.

20.2.2 Le programme de test

```
#USE "course.lib"

INT FUNCTION spg(VAL INT vg, vd)
  VAL INT s IS 0  :
  VAL INT p IS 1  :
  VAL INT g IS 2  :
  VAL []INT t.spg IS [s,s,s,s,p,g,g,g,g]  :
  INT result      :
  VALOF
    result := t.spg[(3*vg) + vd]
    RESULT result
:

PROC racine(CHAN INT fg.in?,fd.in?, fg.out!, fd.out!)
  VAL INT fin IS #FF  :
  VAL INT p   IS 1    :
  VAL INT s   IS 0    :
  INT vg, vd , carry  :
  SEQ
    PAR
      fg.in? vg
      fd.in? vd
    carry := spg(vg, vd)
    IF
      carry = p
        carry := s
      TRUE
        SKIP

    fg.out! vd
    fd.out! p
    --
    PAR
      fg.out!carry
      fd.out!carry
    --
    PAR
      fg.out! fin
      fd.out! fin
:

PROC interne(CHAN INT fg.in?,fd.in?, fg.out!, fd.out!, p.in?, p.out!)
  VAL INT p    IS 1     :
  VAL INT exit IS 4     :
  VAL INT fin  IS #FF   :
  INT vg, vd, code      :
  SEQ
```

```
    PAR
      fg.in ? vg
      fd.in ? vd
    PAR
      fg.out! vd
      fd.out! p

    code := spg(vg, vd)
    p.out ! code
    --  lit le code retourne par le pere
    --  celui de l'homologue gauche si fils droit
    --  le code p si fils gauche
    p.in? code
    --
    -- *********************  phase finale
    --
    WHILE code <> fin
      SEQ
        PAR
          fg.out! code
          fd.out! code
        p.in? code
    PAR
      fg.out! fin
      fd.out! fin
:

PROC feuille(VAL INT pid, x ,y ,  CHAN INT p.in?, p.out!, SHARED CHAN BYTE scr!)
  VAL INT p   IS 1    :
  VAL INT nul IS -1    :
  VAL INT fin IS #FF  :
  INT j, bx, by, code :
  INT v               :
  BYTE s              :
  SEQ
    j := 30 - pid
    --  j varie de 15 a 0
    -- calcul du code initial(s,p,g)
    -- emission de ce code au pere
    IF
      (x BITAND (1 << j)) > 0
        bx := 1
      TRUE
        bx := 0
    IF
      (y BITAND (1 << j)) > 0
        by := 1
      TRUE
        by := 0
    code := bx + by
    p.out! code
   -- le pere a calcule spg et retourne aux 2 fils
    p.in? code
```

```
    --  phase finale
    p.in? v
    WHILE v <> fin
      SEQ
        IF
          (code = p) AND (v <> p)
            code := v
          TRUE
            SKIP
        p.in? v

    IF
      code > 1
        code := 1
      TRUE
        SKIP
    s := (BYTE (((bx+by)+code) REM 2))
    CLAIM scr!
      SEQ
        cursor.x.y((BYTE pid),10, scr!)
        scr! '0' + s
:

PROC main(SHARED CHAN BYTE scr!)
  INT op1, op2               :
  [31]CHAN INT pf , fp       :
  SEQ
    CLAIM scr!
      erase.screen(scr!)
    --  la somme de #5C92 et #685C vaut #C4EE
    op1 := #5C92
    op2 := #685C
    PAR
      racine (fp[1]?,fp[2]?, pf[1]!, pf[2]!)
      PAR i= 1 FOR 14
        interne(fp[(2*i)+1]?, fp[(2*i)+2]?, pf[(2*i)+1]!, pf[(2*i)+2]!, pf[i]?, fp[i]!)
      PAR i= 15 FOR  16
        feuille(i, op1, op2, pf[i]?, fp[i]!, scr!)
    CLAIM scr!
      out.string("*n*nBY...*n",0,scr!)
:
```

A l'exécution l'écriture binaire du résultat est affiché sur le terminal.
Le programme additionne 5C92 et 685C ce qui donne C4EE.

```
              1100010011101110

BY...
```

Chapitre 21

Les outils

21.1 Le compilateur Kroc

Toutes les informations pour se procurer Kroc ainsi que la documentation qui l'accompagne sont regroupées à : `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

La version actuelle de Kroc est la 1.5.0 (en Novembre 2010). On consultera également avec profit le site : `http://pop-users.org/wiki/occam-pi`.
On trouvera dans cette annexe l'installation pour les plateformes Debian et derivées (dont Ubuntu). Les lignes qui suivent ne font que reprendre , pour l'essentiel, les recommandations préconisées pour installer kroc sous Debian.

Note importante : Les versions 8.xx jusqu'à 10.xx de Ubuntu ne posent aucun problème pour l'installer kroc .
Par contre à partir de la version 11.04 de Ubuntu, à l'heure où ces lignes sont écrites, (Janvier 2012) des problèmes existent liés au linker de la nouvelle version de gcc.

21.1.1 Les préréquisites

Avant de télécharger le compilateur proprement dit les packages requis doivent être installés en exécutant (sous root) :

```
sudo apt-get install  libc6-dev libsdl1.2-dev libsdl-sound1.2-dev libgl1-mesa-dev \
  libmysqlclient15-dev libpng12-dev libxmu-dev libxi-dev \
  libplayercore2-dev libplayerc2-dev libltdl3-dev \
  python drscheme xsltproc subversion gcc-multilib
```

21.1.2 Installer kroc

Le système de gestion de version (svn) est maintenant invoqué et permet d'obtenir la dernière version de kroc dans le dossier “kroc-svn”.

```
svn co http://projects.cs.kent.ac.uk/projects/kroc/svn/kroc/branches/kroc-1.5 kroc-svn
```

Le dossier kroc-svn se trouve dans le répertoire où la commande svn a été exécutée.
On se déplace (cd / ..../kroc-svn) dans le répertoire kroc-svn .
En étant root (sudo) on exécute alors :

```
sudo ./build --prefix=/usr/local
```

kroc ainsi que ses librairies seront implantés sous /usr/local qui est le choix de l'auteur.
"- -prefix=repertoire" permet de choisir le repertoire lié à l'implémantation.
Avec le choix qui est fait l'exécutable est dans /usr/local/bin et les librairies sont dans /usr/local/lib.

21.1.3 Faire reconnaitre les librairies de kroc

Pour que l'éditeur de lien fasse correctement son travail il est nécessaire que la variable d'environnement LD_LIBRARY_PATH soit affectée à la valeur : "/usr/local/lib".
Pour ce faire on édite le fichier (caché) .bashrc en y insérant les deux lignes suivantes :

```
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH
```

On actualise .bashrc sans recharger le système en exécutant : **source** .bashrc . Il est recommandé d'exécuter la commande **printenv** pour s'assurer que la variable LD_LIBRARY_PATH est bien reconnue.

21.1.4 La compilation séparée

La philosophie de kroc relativement à la compilation séparée s'inspire de celle du compilateur gcc bien qu'elle fasse usage de fichiers ***.tce** qui, pour l'essentiel,contiennent du byte code pour la machine virtuelle et des informations sur les signatures des processus nommés et des fonctions déclarés dans le fichier texte soumis à la compilation.
Les fichiers ***.o** contiennent du code natif.
Un fichier **qui ne contient pas** le texte de main() se compile séparemment avec l'option **-c** et donne naissance à un fichier "objet" **.o** et un fichier **.tce**.
Considérons le fichier texte addquatre.occ :

```
PROC add.quatre(INT x)
  x := x+4
:
```

La commande "kroc addquatre.occ -c" donne naissance aux fichiers addquatre.o et addquatre.tce situés dans le même répertoire que addquatre.occ. Considérons maintenant le fichier cadd.occ situé dans le même répertoire que addquatre.o :

```
#USE "addquatre"
#USE "course.lib"

PROC main(CHAN BYTE kbd?, scr!)
  INT i :
  SEQ
    i := 0
    add.quatre(i)
    out.int(i,0, scr!)
    out.string("*nBY ...*n",0, scr!)
:
```

La directive “#USE addquatre” fait référence au fichier addquatre.tce. Le fichier cadd.occ se compile avec la commande :

```
kroc cadd.occ addquatre.o -lcourse
```

A l'exécution 4 s'affiche à l'écran montrant ainsi la prise en compte du processus nommé add.quatre().

21.1.5 Ecrire ses propres librairies

Soient les trois fichiers a.occ,b.occ et c.occ contenant respectivement les textes des processus nommés a(), b() et c() :

```
-- a.occ

PROC a(BYTE i)
  i := i + 'a'
:
-- b.occ
--

PROC b(BYTE i)
  i := i + 'a'
:
-- c.occ
--

PROC c(BYTE i)
  i := i + 'c'
:
```

On compile séparemment ces trois fichiers comme il a été dit ci dessus.
Il en résulte trois fichiers .o et trois fichiers .tce.
On regroupe les trois fichiers .tce en un seul fichier “abc.lib” en exécutant :

```
ilibr a.tce b.tce c.tce -o abc.lib
```

De la même façon on regroupe les trois fichiers .o en un seul fichier objet “liboccam_abc.so” en exécutant :

```
ld -r -o liboccam_abc.so a.o b.o c.o
```

Soit maintenant le fichier test.occ :

```
#USE "abc.lib"

PROC main(CHAN BYTE kbd?, scr!)
  BYTE bt :
  SEQ
    bt := 0
    a(bt)
```

```
    scr! bt
    scr! '*n'
    --
    bt := 0
    b(bt)
    scr! bt
    scr! '*n'
    --
    bt := 0
    c(bt)
    scr! bt
    scr! '*n'
:
```

On remarque la référence à "#USE abc.lib" qui renvoie aux trois fichiers .tce combinés en un seul fichier abc.lib.
La compilation de test.occ et sa liaison à abc.lib et liboccam_abc.so se fait en exécutant :

```
kroc test.occ  -labc
```

Le fichier liboccam_abc.so a été utilisé à l'édition de liens bien qu'il n'apparaisse pas dans la commande. Ceci tient au fait qu'il est situé dans le même répertoire que le fichier test.occ.
Dans le cas général les fichiers .lib sont regroupés dans un même répertoire.
L'éditeur de lien utilise alors la variable d'environnement OCSEARCH qui fait référence à ce répertoire. Par exemple si le repertoire "liboccam" situé dans le répertoire d'accueil "accueil" contient abc.lib on édite ,(comme pour LD_LIBRARY_PATH) le fichier .bashrc en lui insérant les deux lignes :

```
OCSEARCH=/home/accueil/liboccam
export OCSEARCH
```

On n'oublie pas d'exécuter la commande : **source** .bashrc .

Pour ce qui est du fichier .so ou bien il réside dans le répertoire ou se trouve le fichier devant être compilé (le fichier qui contient le texte du processus main()), ou bien il est copié dans le répertoire où réside les librairies de kroc (dans le cas présent /usr/local/lib).
La compilation et l'édition de liens se fait toujours par :

```
kroc test.occ  -labc
```

21.2 L'éditeur Kate

Les sources Occam peuvent être éditées par tout éditeur de texte.
Néanmoins un éditeur repliable (folding editor) est recommandé car la syntaxe d'Occam étant très gourmande en texte il peut être nécessaire de pouvoir en replier des portions importantes pour gagner en lisibilité.
Je recommande plus particulièrement **Kate** qui, bien que destiné à l'environnement KDE, s'installe aisémment sous GNOME.

Kate est facilement paramétrable pour la coloration syntaxique d'occam-π . Il suffit de télécharger le fichier xml dédié a occam-π disponible dans la rubrique Text Editors des ressources fournies à l'adresse :
`http://pop-users.org/wiki/occam-pi/LearningResources`.

On dézippe le fichier kate-occ-pi-syntax.zip pour obtenir le fichier occam-pi.xml.
Ensuite, étant root, il suffit de copier ce fichier dans le répertoire
.kdedir/apps/katepart/syntax qui contient tous les fichiers xml de configuration pour les langages proposés en standart par Kate. (kdedir est le dossier relatif à KDE (KDE3 ou KDE4) dans le système de fichiers.
Sous (K)Ubuntu /usr/share/kde4/apps/katepart/syntax est le répertoire approprié.
Pour terminer, on peut directement invoquer Kroc depuis la console intégrée à Kate qui se synchronise automatiquement sur le dossier contenant le fichier texte en cours de traitement (le demander dans la configuration de Kate).
NB : Pour éviter les conflits potentiels entre les distributions de Kate et les différentes versions de GNOME il est plus prudent de travailler sous KUBUNTU pour lequel KDE est l'environnement standart.

Chapitre 22

Les modules de kroc

```
* Module button  - A Button library that uses OpenGL and SDL
* Module cif  - C interface to the scheduler
* Module convert
* Module course  - Course library
* Module course.cycles  - Demonstration cycles
* Module course.networks  - Demonstration networks
* Module file  - File library
* Module fmtout  - Formatted output utilities
* Module forall  - Definitions available to all occam-pi programs
* Module g3dchess.inc
* Module graphics3d  - A library for programming simple 3D graphics in occam-pi
* Module hostio
* Module hostsp
* Module maths  - Mathematical functions and constants
* Module occGL  - Interface to foreign library occGL
* Module occSDL  - Interface to foreign library occSDL
* Module occSDLsound  - Interface to foreign library occSDLsound
* Module occade  - A library for programming simple arcade games in occam-pi
* Module occplayer  - Interface to foreign library occplayer
* Module overwriting-buffer - Generic N-place overwriting buffer
* Module player  - Higher-level bindings to the Player library
* Module pony  - Transparent networking library for occam-pi programs
* Module proc  - Process library
* Module random  - Random number generation
* Module raster  - Basic raster graphics support
* Module rastergraphics  - Graphics primitives for rasters
* Module rasterio  - Support for saving and loading rasters as graphics files
* Module rastertext  - Support for drawing text onto a raster
* Module sdlraster  - SDL raster interface
* Module selector  - IO selector
* Module shared_screen  - Shared screen library
* Module sock  - Socket library
* Module streamio
* Module string
```

```
* Module time  - Time utilities
* Module trap  - An asynchronous network messaging system for occam-pi
* Module ttyutil  - A utility library for terminal-based input and output
* Module tvm.specials  - Include file for the transterpreter's magical C hooks
* Module udc  - User-defined channels
* Module useful  - General utilities
* Module video  - V4L(2) frame-grabber interface
```

La description précise de ces modules se trouve à : `http://occam-pi.org/occamdoc/`

Bibliographie

[dR94] J de Rumeur. *Communication dans les réseaux de processeurs.* Etudes et recherches en informatique. Masson, 1994.

[eDW01] Davide Sangiorgi et David Walker. *The Pi-Calculus :A théory of Mobile Processes.* Cambridge University Press, 2001.

[ePW04] Fred Barnes et Peter Welch. Communicating mobile processes. 2004. Disponible en ligne a : //ftp.cs.ukc.ac.uk/pub/phw/sei-cmu/occam/papers/.

[Hoa04] C.A.R Hoare. *Communicating Sequential Processes.* Prentice Hall, 2004. Disponible en ligne à : http ://www.usingcsp.com/cspbook.pdf.

[Lei95] F.Thomson Leighton. *Introduction aux algorithmes et architectures distribuées.* Morgan Kaufmann, 1995.

[Lim88] Inmos Limited. *Occam2 Reference Manual.* C.A.R Hoare Series Editor. Prentice Hall, 1988.

[Mil89] Robin Milner. *Communication and Concurrency.* C.A.R Hoare Series Editor. Prentice Hall, 1989.

[Mil07] Robin Milner. *Communicating and mobile systems :the pi-calculus.* Cambridge University Press, 2007.

[MPW92] Robin Milner, Joakim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, pages 41–77, 1992. Disponible en ligne à : http ://move.to/mobility.

[Par95] Joachim Parrow. Interaction diagrams. an easy introduction to mobile processes. *Nordic Journal of Computing*, pages 407–443, 1995. Disponible en ligne à : http ://move.to/mobility.

[Par01] Joachim Parrow. *An Introduction to the pi-calculus*, volume HandBook of Process Algebra, pages 479–543. Elsevier, 2001. Disponible en ligne à : http ://move.to/mobility.

[Ray85] Michel Raynal. *Algorithmes distribués et protocoles.* Eyrolles, 1985.

[Ros98] A.W Roscoe. *The théory and practice of concurrency.* Prentice Hall, 1998. Disponible en ligne à : http ://web.comlab.ox.uk/people/Bill.Roscoe/publications/68b.pdf.

[Tho95] SGS Thomson. *Occam 2.1 Reference Manual.* SGS-Thomson Microelectronics, 1995. Disponible en ligne à : http ://www.wotug.org/occam/documentation/oc21refman.pdf.

Index

Ackermann, 90
affectation, 40
AFTER, 31, 49, 74
ALT, 67
alternative, 67
AND, 30
asynchrone, 74

barrière, 50
barrière mobile, 152
barrière partielle, 114
BITAND, 28
BITNOT, 28
BITOR, 28
bloc CLAIM, 100
bnf, 3
bouts d'un canal mobile, 117

canal mobile, 117
canal partagé, 99
canal virtuel, 202
CASE, 57
cast, 27
champs, 22
CHAN TYPE, 117
chien de garde, 72
codage de canaux, 210
communication, 40
constante, 23
CSP, iv

data.struct, 11
data.table, 10
David Walker, iv
declarations, 5
diffusion, 223
directives, 5
données mobiles, 111

ELSE, 57

FDR, iv
flit, 182, 202
fonctions, 92
FOR, 54
FROM, 20
FUNCTION, 92

graphe complet, 227
grille torique, 181
guarde, 67
Guillaume d'Occam, iii

IF, 55
INCLUDE, 5
indentation, 3
INMOS, iii
IS, 11

Joakim Parrow, iv

kate, 246
kroc, 243

main, 6
MINUS, 29
MOBILE, 111
MOSTNEG, 30
MOSTPOS, 30
multiplexage, 70, 202

NOT, 30

Occam2.1, iv
opérateurs arithmétiques, 28
opérateurs booléens, 30
opérateurs de décalage, 28
oprateurs relationnels, 30
OR, 30

P.H Welch, 65
PAR, 61
parallélisme des liens, 188, 215
parcourt d'arbre, 91
Peter Welch, iv
pi-calculus, iv
pipe line, 82
PLUS, 29
PRI ALT, 67
pri par, 65

priorité des opérateurs, 27
PROC, 77
PROC TYPE, 132
processeurs multi-coeurs, iii
processus, 39
processus alternatif, 67
processus alternatif repliqué, 69
processus boucle, 58
processus conditionnel, 55
processus conditionnel repliqué, 56
processus nommé, 77
processus parallèle, 61
processus parallele replique, 64
processus récursif, 90
processus séquentiels, 53
processus sequentiel repliqué, 54
programme occam, 5
protocole, 40
protocole sequentiel, 12, 47
protocole variable, 13

REC PROC, 90
REM, 28
rendez vous, 40
renommer un type, 11
RESULT, 80, 92
Robin Milner, iv
ROUND, 32
routage dans l'hypercube, 211
routage dans la grille, 195
routage dans la grille torique , 181
routage store and forward, 182
routage whorm hole, 202

sélection, 57
segmen, 20
SEQ, 53
SHARED, 99
SIZE, 31
SKIP, 52
STEP, 54
STOP, 52
structure, 17
suspension d'un processus mobile, 133
SYNC, 50

tableau, 17
tag, 13
TIMES, 29
Tony Hoare, iv
Transputer, iii, iv
trie, 191
TRUNC, 32

USE, 5

VAL, 23

WHILE, 58