# Informatique Parallèle et Distribution

Eric Goubault
Commissariat à l'Energie Atomique

# Table des matières

# Chapitre 1

# Introduction

Le but de ce cours est de donner une idée de la programmation, algorithmique et sémantique de systèmes parallèles et distribués; vaste tâche! Pour y arriver, on utilise donc un biais, qui nous permettra d'aborder la plupart des thèmes ultra-classiques du domaine. On va utiliser le système de "threads" de JAVA pour simuler des processus parallèles et distribués.

Les "threads" en tant que tel sont une première approche du parallélisme qui peut même aller jusqu'au "vrai parallélisme" sur une machine multi-processeurs. Pour avoir un panorama un peu plus complet du parallélisme et de la distribution, il faut être capable de faire travailler plusieurs machines (éventuellement très éloignées l'une de l'autre; penser à internet) en même temps. On en fera une approche, soit par simulation par des threads, soit par le mécanisme de sockets dans le chapitre 6, qui arrivera plus tard...

Dans les sections 3.2.1 et 3.2.3, on présente d'abord intuitivement le modèle de threads de JAVA (communication par mémoire partagée) et la façon dont on peut créer et contrôler un minimum les threads utilisateurs. On voit un peu plus en détail à la section 3.2.2 le fonctionnement de la JVM en présence de threads. On donne des éléments plus avancés de contrôle des threads JAVA à la section 3.3, en particulier en ce qui concerne le cycle de vie des threads, les groupes de processus, les priorités, et l'ordonnancement.

Dans le chapitre suivant on s'intéresse au très classique et important problème de l'accès concurrent aux variables partagées. En effet la machine JVM sous-jacente n'est pas une P-RAM CRCW mais ressemble plus à une CREW! (voir Chapitre 1 de [Rob00]) Il faut donc faire très attention à verrouiller les accès, au moins au moment de l'écriture comme on le verra à la section 4.1. Pour ce faire on utilisera essentiellement le mot clé `synchronized` introduit à la section 4.2. On en profitera pour voir un peu de sémantique très minimaliste dans la section 4.4.2. En effet le premier problème que l'on rencontre, et que l'on ne voit pas dans le monde séquentiel, est le problème de point mort ou étreinte fatale, ou encore deadlock en anglais. On commence à le rencontrer en section 4.4.2 mais ce sera toujours un thème récurrent.

A partir de `synchronized` on peut implémenter les mécanismes de synchronisation et d'exclusion mutuelle typiques que l'on rencontre sous une forme ou une autre dans tout langage parallèle. On commence par les moniteurs [Hoa74], section 4.3, puis les sémaphores binaires [Dij68], section 4.4.1 et les sémaphores plus généraux, dits à compteur, section 4.4.5, qui sont d'ailleurs implémentables à partir des sémaphores binaires [Dij68]. En fait, `synchronized` n'est jamais qu'un sémaphore binaire (ou verrou, ou mutex) associé à une méthode ou à un bout de code. On en profite aussi dans la section 4.5 pour donner une implémentation des barrières de synchronisation, très utilisées dans des programmes parallèles à gros grain, par exemple fonctionnant sur *cluster* de PC.

Quand on ne veut pas mettre des barrières de synchronisation partout, mais que l'on veut ordonnancer ces processus plus mollement (on peut alors gagner en efficacité car l'ordonnancement a plus de degrés de liberté) de façon à ce que l'état global du système reste “cohérent” en un certain sens, on tombe sur les problèmes classique de séquentialisation. On en donne quelques exemples, très classiques en bases de données distribuées, et on développe le protocole le plus connu permettant d'assurer cette cohérence dans ce cadre, le protocole 2PL (“2-Phase Locking”) ou protocole à deux phases, en section 4.6. On se reportera au cours de S. Abiteboul, [Abi00], pour plus de détails et de références.

On voit que l'on se rapproche de plus en plus de la distribution. C'est bien le but et on trouve dans le chapitre 6. une façon de simuler le passage de messages entre processus par la mémoire partagée de JAVA. Dans la section suivante on dit un mot de la façon de programmer sur réseau de stations UNIX, par les sockets. Cela reste très bref car c'est traité dans le cours de F. Bourdoncle [Bou00]. Ensuite on fait quelques classiques de la programmation distribuée, exclusion mutuelle, élection, estampillage, dans le chapitre 6

Entre temps, on en aura profité au le chapitre 5 pour voir les grands classiques des algorithmes permettant de programmer l'exclusion mutuelle (le `synchronized`) sur des machines à mémoire partagée et sous certaines hypothèses. Cela peut paraître assez académique mais est un excellent moyen de s'habituer à raisonner sur des programmes parallèles se coordonnant par lecture écriture en mémoire partagée, et ce n'est pas si facile! Cela nous permet également d'introduire à la preuve de programmes.

La dernière section est consacrée au problème de la programmation distribuée en présence de pannes (crash) d'une on plusieurs machines. On se contente de traiter du cas très classique du consensus en présence d'une panne dans un système à mémoire partagée (avec lecture et écritures atomiques). C'est également un domaine considérable, que l'on ne fait qu'aborder et on recommande au lecteur de se plonger dans la lecture de [Lyn96] pour avoir quelques compléments.

**Références** Pour des compléments, ou des applications du parallélisme au calcul scientifique, on pourra se reporter à [GNS00]. Ces notes très préliminaires sont un complément du cours de Yves Robert [Rob00] qui montre l'algorithmique et la complexité moins proches de la machine. On se reportera aussi au complément de D. Rémy [Rem00] pour tout ce qui concerne l'orienté objet. Le livre [Ray97] apportera au lecteur toutes les précisions voulues sur les algorithmes vus au chapitre 5.Enfin, je remercie Matthieu Martel d'avoir relu cette première version de polycopié. Les erreurs qui restent (nombreuses!) sont néanmoins les miennes!

# Chapitre 2

# Introduction au parallélisme

Ce chapitre est bien sûr loin d'être exhaustif. On n'y parle pratiquement pas de micro-parallélisme, c'est à dire du parallélisme au niveau des microprocesseurs. En effet la plupart des microprocesseurs modernes sont à eux tous seuls de véritables machines parallèles. Par exemple il n'est pas rare (par exemple Pentium) d'avoir plusieurs unités de calcul arithmétique (dans le cas du Pentium, d'unités de calcul flottant, ou dans le cas des processeurs MIPS, plusieurs additionneurs, multiplicateurs etc.) pouvant fonctionner en parallèle. On parle un peu plus loin du calcul en pipeline qui est courant dans ce genre d'architectures.

A l'autre bout du spectre, on ne parle que très des projets les plus en vogue à l'heure actuelle, c'est à dire des immenses réseaux d'ordinateurs hétérogènes distribués à l'echelle d'internet. On pourra se reporter aux divers projets de "metacomputing", voir par exemple `http://www.metacomputing.org/` et au projet "GRID", voir par exemple `http://www.gridforum.org/`.

## 2.1 Une classification des machines parallèles

Une machine **parallèle** est essentiellement un ensemble de **processeurs** qui **coopèrent** et **communiquent**.

Historiquement, les premières machines parallèles sont des **réseaux** d'ordinateurs, et des machines **vectorielles** et faiblement parallèles (années 70 - IBM 360-90 vectoriel, IRIS 80 triprocesseurs, CRAY 1 vectoriel...).

On distingue classiquement quatre types principaux de parallélisme (Taxonomie de **Tanenbaum**): **SISD**, **SIMD**, **MISD** et **MIMD**. Même si de nos jours cette classification peut paraître un peu artificielle (le moindre micro-processeur courant inclut lui-même plusieurs formes de micro-parallélisme). Cette classification est basée sur les notions de flot de **contrôle** (deux premières lettres, I voulant dire "Instruction") et flot de **données** (deux dernières lettres, D voulant dire "Data").

### 2.1.1 Machine SISD

Une machine **SISD** (**S**ingle **I**nstruction **S**ingle **D**ata) est ce que l'on appelle d'habitude une machine de Von Neuman. Une seule instruction est exécutée à un moment donné et une seule donnée (simple, non-structurée) est traitée à un moment donné.

Le code suivant,

```
int A[100];
...
for (i=1;100>i;i++)
```

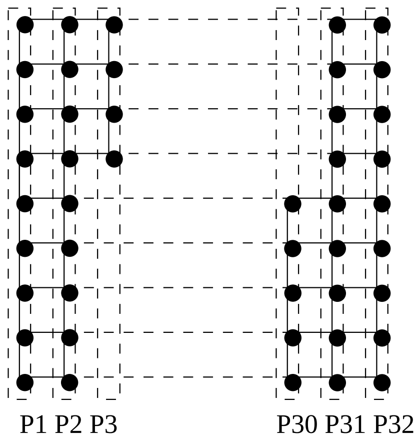



FIG. 2.1 – *Répartion d'un tableau sur les processeurs d'une machine SIMD typique*

```
A[i]=A[i]+A[i+1];
```

s'exécute sur une machine séquentielle en faisant les additions `A[1]+A[2]`, `A[2]+A[3]`, etc., `A[99]+A[100]` à la suite les unes des autres.

### 2.1.2 Machine SIMD

Une machine **SIMD** (**S**ingle **I**nstruction **M**ultiple **D**ata) est une machine qui exécute à tout instant une seule instruction, mais qui agit en parallèle sur plusieurs données, on parle en général de "parallélisme de données". Les machines SIMD peuvent être de plusieurs types:

(1) Parallèle
(2) Systolique

Les machines systoliques sont des machines SIMD particulières dans lesquelles le calcul se déplace sur une topologie de processeurs, comme un front d'onde, et acquière des données locales différentes à chaque déplacement du front d'onde (comportant plusieurs processeurs, mais pas tous en général comme dans le cas (1)).

En général dans les deux cas, l'exécution en parallèle de la même instruction se fait en même temps sur des processeurs différents (parallélisme de donnée **synchrone**). Examinons par exemple le code suivant (cas (1)) écrit en CM-Fortran sur la Connection Machine-5 avec 32 processeurs,

```
      INTEGER I,A(32,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL)
      ...
      FORALL (I=1:32,J=1:1000)
     $   A(I:I,J:J)=A(I:I,J:J)+A(I:I,(J+1):(J+1))
```

Chaque processeur $i$, $1 \leq i \leq 32$ a en sa mémoire locale une tranche du tableau `A`: `A(i,1), A(i,2), ..., A(i,1000)`. Il n'y a pas d'interférence dans le calcul de la boucle entre les différentes tranches: tous les processeurs exécutent la même boucle sur leur propre tranche en même temps (cf. figure 2.1).

### 2.1.3 Machine MISD

Une machine **MISD** (**M**ultiple **I**nstruction **S**ingle **D**ata) peut exécuter plusieurs instructions en même temps sur la même donnée. Cela peut paraître paradoxal mais cela recouvre en fait un type très ordinaire de micro-parallélisme dans les micro-processeurs modernes: les processeurs vectoriels et les architectures pipelines.

Un exemple de “pipelinage” d'une addition vectorielle est le suivant. Considérons le code:

```
FOR  i:=1 to n DO
   R(a+b*i):=A(a'+b'*i)+B(a''+b''*i);
```

`A`, `B` et `R` sont placés dans des registres vectoriels qui se remplissent au fur et à mesure du calcul,

| Temps | $A$ | ( | $i$ | ) | $B$ | ( | $i$ | ) | $R$ | ( | $i$ | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | . | . | . | 1 | . | . | . | . | . | . | . |
| 2 | 2 | 1 | . | . | 2 | 1 | . | . | . | . | . | . |
| 3 | 3 | 2 | 1 | . | 3 | 2 | 1 | . | . | . | . | . |
| 4 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | . | . | . | . |
| 5 | 5 | 4 | 3 | 2 | 5 | 4 | 3 | 2 | 1 | . | . | . |
| 6 | 6 | 5 | 4 | 3 | 6 | 5 | 4 | 3 | 2 | 1 | . | . |
| *etc.* | | | | | | | | | | | | |

En ce sens, quand le pipeline est rempli, plusieurs instructions sont exécutées sur la même donnée.

### 2.1.4 Machine MIMD

Le cas des machines **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) est le plus intuitif et est celui qui va nous intéresser le plus dans ce cours. On a plusieurs types d'architecture possibles:

(1) Mémoire partagée (Sequent etc.)

(2) Mémoire locale + réseau de communication (Transputer, Connection Machine, local, par réseau d'interconnexion) - Système réparti

C'est le cas (1) que l'on va voir plus particulièrement avec JAVA. On pourra également simuler le cas (2). Pour le cas (2) (en PVM et MPI) et en particulier des applications au calcul scientifique, on pourra se reporter par exemple au cours [GNS00].

**Mémoire Partagée:**

Une machine MIMD à mémoire partagée est principalement constituée de processeurs avec des horloges indépendantes, donc évoluant de façon asynchrone, et communicant en écrivant et lisant des valeurs dans une seule et même mémoire (la mémoire partagée). Une difficulté supplémentaire, que l'on ne décrira pas plus ici, est que chaque processeur a en général au moins un cache de données (voir figure 2.2), tous ces caches devant avoir des informations cohérentes aux moments cruciaux.

La synchronisation des exécutions des processeurs (ou processus, qui en est une “abstraction logique”) est nécessaire dans certains cas. Si elle n'était pas faite, il y aurait un risque d'**incohérence** des données. Partant de $x = 0$, exécutons $x := x+x$

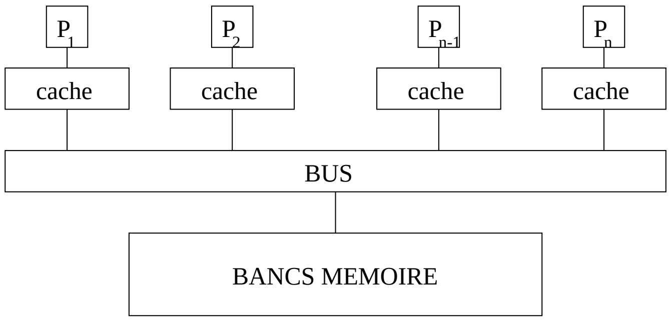



FIG. 2.2 – *Architecture simplifiée d'une machine à mémoire partagée*

en parallèle avec $x := 1$. On a alors essentiellement trois exécutions possibles (en supposant chacune des affectations compilées en instructions élémentaires insécables, ou "*atomiques*", comme suit):

| | | |
|---|---|---|
| `LOAD x,R1` | `WRITE x,1` | `LOAD x,R1` |
| `LOAD x,R2` | `LOAD x,R1` | `WRITE x,1` |
| `WRITE x,1` | `LOAD x,R2` | `LOAD x,R2` |
| `WRITE x,R1+R2` | `WRITE x,R1+R2` | `WRITE x,R1+R2` |
| Résultat x=0 | Résultat x=2 | Résultat x=1 |

Cela n'est évidemment pas très satisfaisant; il faut rajouter des synchronisations pour choisir tel ou tel comportement. Cela est en particulier traité à la section 4.6.

La synchronisation peut se faire par différents mécanismes:

- Barrières de synchronisation,
- Sémaphores : deux opérations $P$ et $V$.
- Verrou (mutex lock) : sémaphore binaire qui sert à protéger une section critique.
- Moniteurs : construction de haut niveau, verrou implicite.
- etc.

Les opérations $P$ et $V$ sur les sémaphores sont parmi les plus classiques et élémentaires (on les verra par la suite en long, en large et en travers, en particulier à la section 4.4). On peut considérer qu'à chaque variable partagée $x$ dans la mémoire est associé un "verrou" (du même nom) indiquant si un processus est en train de la manipuler, et ainsi en en interdisant l'accès pendant ce temps. L'opération $Px$ exécutée par un processus verrouille ainsi son accès exclusif à $x$. L'opération $Vx$ ouvre le verrou et permet à d'autres processus de manipuler $x$ à leur tour.

La encore des erreurs sont possibles, en voulant trop synchroniser par exemple. Il peut y avoir des cas d'*interblocage* (deadlock, livelock) en particulier.

Supposons qu'un processus $T_1$ ait besoin (pour effectuer un calcul donné) de verrouiller puis déverrouiller deux variables $x$ et $y$ dans l'ordre suivant: $Px$ puis $Py$ puis $Vx$ puis $Vy$ alors qu'un autre processus, en parallèle, désire faire la séquence $Py$, $Px$ puis $Vy$ et enfin $Vx$. En fait les deux processus peuvent s'interbloquer l'un l'autre si $T_1$ acquiert $x$ (respectivement $T_2$ acquiert $y$) puis attend $y$ (respectivement $x$). Tout cela sera encore traité à la section 4.4.2.

### Machine distribuée

En général, l'emploi d'autres mécanismes de communication que la mémoire partagée pour une machine MIMD est due au fait que les processeurs sont physiquement trop éloignés pour qu'un partage de petites informations soit raisonnable, par exemple, le réseau Internet permet de considérer en un certain sens, tous les

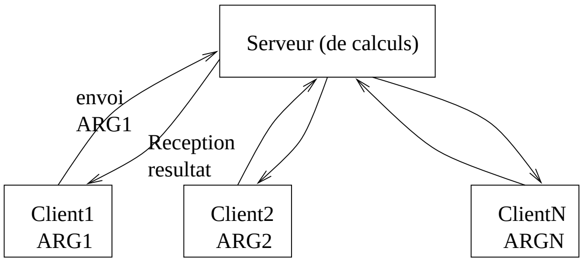



FIG. 2.3 – *Architecture Client/Serveur (RPC)*

ordinateurs reliés comme étant un seul et même ordinateur distribué, où les processeurs travaillent de façon asynchrone et où les informations transitent par passage de message. Ce n'est en fait pas toujours le cas, et un certain nombre de supercalculateurs travaillent par échange de messages également.

De façon générale, la synchronisation et l'échange d'information peuvent se faire par,

- Appel de procédure distribuée (RPC) :
  - réponse synchrone
  - réponse asynchrone
- Envoi/Réception de message asynchrone (tampon de taille limitée ou non); active polling ou gestion à l'arrivée par une procédure `handler`.
- Envoi/Réception de message synchrone : rendez-vous.
- Mélange des deux derniers cas.

Le protocole RPC ("Remote Procedure Call") entre machines UNIX avec réseau Ethernet est un exemple de programmation MIMD (cf. figure 2.3).

### 2.1.5 Remarque

Les architectures sont plus ou moins bien adaptées à certains problèmes. En fait, le gain de temps espéré, qui est d'au plus $N$ (nombre de processeurs)est rarement atteint et, en pratique, parallélisme difficile à contrôler. Enfin, pour revenir à notre première remarque sur la taxonomie de Tanenbaum, il y a de plus en plus rarement une distinction tranchée entre le modèle mémoire partagée et celui par passage de messages; dans la realité, la situation est très hybride.

## 2.2 Contrôle d'une machine parallèle

Il y a deux méthodes principales,

(1) On dispose d'un language séquentiel: le compilateur **parallélise** (Fortran...),

(2) On a une extension d'un langage séquentiel ou un langage dédié avec des constructions parallèles **explicites** (Parallel C, Occam...)

On pourra se reporter à [Rob00] et [GNS00] pour les techniques dans le cas (1). On se concentre ici principalement sur le deuxième cas. Il faut alors disposer de primitives (ou instructions) pour:

- La création de parallélisme,
  - Itération simultanée sur tous les processeurs (**FOR** parallèle de Fortran ou **FORALL**),

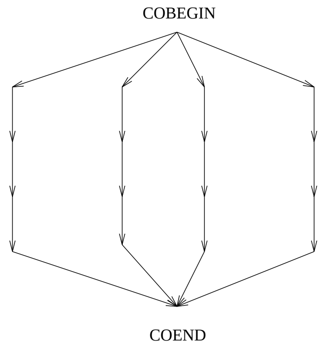



FIG. 2.4 – *Représentation simplifiée du flot d'exécution dans le cas d'un* **COBEGIN/COEND**

  - Définition d'un ensemble de **processus** (**COBEGIN** cf. figure 2.4),
  - Création de **processus** (**FORK** de Unix)
- Contrôler les tâches parallèles,
  - Synchronisation (**Rendez-vous** d'Ada),
  - Passage de messages (synchrones/asynchrones, **broadcast**,...),
  - Section critiques (gestion de la mémoire partagée),
  - Arrêt d'une exécution parallèle (**COEND**, **WAIT** d'Unix)

## 2.3 Quelques exemples de machines parallèles

Pour rêver un peu, on décrit ci-après quelques architectures typiques de supercalculateurs. Il est également intéressant de se reporter au chapitre traitant du routage dans le cours d'Y. Robert [Rob00].

### 2.3.1 La Connection Machine-5

Elle est composée de,

- 32 à 65536 noeuds qui chacun comporte,
  - un processeur Sparc (32 MHz, 22 Mips, 5 MFlops)
  - 4 unités vectorielles (32 MFlops chacune) pour les opérations flottantes
  - 4 bancs mémoire de 8 Mo de DRAM
- Réseau d'interconnexion en arbre gras (hypercube), entre voisins on a: bande passante: 20 Mo/s, latence: 5 $\mu$s
- Entrées/Sorties en parallèle à un système de disques, bande passante: 33 Mo/s, latence: 75 ms

Plusieurs modèles (voir figure 2.5) ont été installés en France, dont un à l'ETCA (32 noeuds, 2 processeurs de contrôle, 1 Go de RAM, 16 Go de disques, 4 GFlops crête, mai 1993).

Elle peut fonctionner en,

- mode SIMD: CM Fortran (sous-ensemble de Fortran 90 et instructions spécifiques d'organisation des données en parallèle), C*
- mode MIMD: de même mais en utilisant la librairie CMMD de passage de messages (on retrouve des ordres similaires dans PVM)

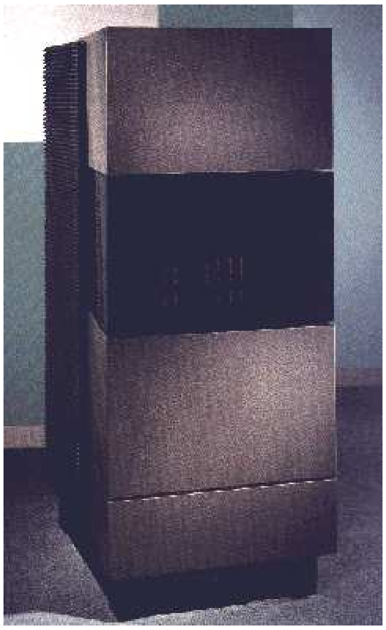


FIG. 2.5 – *Une Connection-Machine 5*

### 2.3.2 Cray T3E

La machine la plus puissante chez CRAY, deux types de modèles, refroidissement à air (pas plus de 128 processeurs) ou par un liquide (jusqu'à 2048 processeurs). Pour les deux types de modèles on peut avoir des configurations (voir figures 2.6 et 2.7):

- De 16 à 2048 noeuds comprenant chacun,
  - un DEC Alpha EV5 (600 MFlops)
  - de 64 Mo à 2 Go de mémoire vive
- Un réseau d'interconnexion en tore 3D (bande passante de 2 à 128 Go/s). Entrées sorties par un réseau en double anneau (allant dans des sens inverses), "GigaRing"

Performance de crête de 9.6 GFlops à 1.2 TFlops, jusqu'à 4 To de mémoire vive. Il peut être utilisé,

- langages: Fortran 90, C et C++ (compilateurs optimisants)
- mode MIMD en utilisant la "Cray Message Passing Toolkit" supportant PVM, MPI (passage de messages) ou la "Shared Memory Library" (le CRAY T3E est organisé logiquement en mémoire partagée même si physiquement ce n'est pas le cas)
- mode SIMD en utilisant Fortran 90, CRAFT ou HPF

### 2.3.3 IBM SP2

L'IBM RISC/6000 Scalable POWERparallel System (SP) est constitué de (voir figure 2.8):

- Noeuds avec un processeur RS/6000 soit à 120 MHz ("thin node" – 480 MFlops) avec 256Mo de mémoire vive et 1,8 Go de mémoire disque, soit à 135 MHz ("wide node" – 540 MFlops) avec 1 à 2 Go de mémoire vive.

FIG. 2.6 – *Un Cray T3E*

FIG. 2.7 – *Un T3E à plusieurs racks*

- Communication par passage de messages à travers un réseau cross-bar à deux niveaux; bande passante de 150Mo par seconde.
- Fonctionne en MIMD avec PVM ou MPI
- Fonctionne en SIMD avec HPF (High Performance Fortran)

### 2.3.4 Réseaux de stations

Un réseau de stations fonctionne en mode MIMD avec une communication par passage de messages (typiquement gérés par PVM ou par JAVA) sur un réseau local ou d'interconnexion.

En fait, un réseau de stations est souvent une architecture hybride dans le sens où il peut y avoir combinaison sur un réseau local de stations et de machines multiprocesseurs (elles-même gérant leur parallélisme interne par mémoire partagée ou par bus ou réseau). JAVA et PVM peuvent également gérer ce genre d'architecture.

### 2.3.5 Top 500

Il est établi régulièrement une liste des 500 machines les plus puissantes, la puissance étant mesurée par un benchmark d'algèbre linéaire et Rmax étant la performance en Gflop/s pour le problème le plus gros programmé sur l'ordinateur (voir `http://www.top500.org`). En novembre 2001 on trouvait:

FIG. 2.8 – *Un IBM SP2*

| Rang | Constructeur | Ordinateur | Rmax |
|---|---|---|---|
| 1 | IBM | ASCI White (USA) | 7226 |
| 2 | Compaq | AlphaServer SC ES45/1 GHz (USA) | 4059 |
| 3 | IBM | SP Power3 375 MHz 16 way (USA) | 3052 |
| 4 | Intel | ASCI Red (USA) | 2379 |
| 5 | IBM | ASCI Blue-Pacific SST,IBM SP 604e (USA) | 2144 |
| ... | ... | ... | ... |
| 9 | IBM | SP Power3 375 MHz 16 way (All) | 1293 |
| ... | ... | ... | ... |
| 56 | IBM | SP Power3 375 MHz (Fr) | 494 |

Pour avoir une idée des puissances requises selon les domaines d'application, on pourra se reporter à [CS98]. En quelques mots, il faut savoir que la prévision météorologique à deux jours demande environ 100 Mégaoctets de mémoire et 100 Mégaflops, pour 3 jours il faut compter environ 1 Gigaoctet pour environ 10 Gigaflops. Le design d'une molécule pharmaceutique nécessite à l'heure actuelle environ 10 Gigaoctets pour 100 Gigaflops.

# Chapitre 3

# Threads Java

## 3.1 Introduction aux threads

Les “threads” ou “processus légers” sont des unités d'exécution autonomes qui peuvent effectuer des tâches, en parallèle avec d'autres threads: ils sont constitués d'un identificateur, d'un compteur de programme, d'une pile et d'un ensemble de registres. Le flot de contrôle d'un thread est donc purement séquentiel. Plusieurs threads peuvent être associés à un “processus lourd” (qui possède donc un flot de contrôle multiple, ou parallèle). Tous les threads associés à un processus lourd ont en commun un certain nombre de ressources, telles que: une partie du code à exécuter, une partie des données, des fichiers ouverts et des signaux.

En Java, le processus lourd sera la JVM (Java Parallel Machine) qui interprète le bytecode des différents processus légers. Les threads coopèrent entre eux en échangeant des valeurs par la mémoire commune (du processus lourd). On en verra plus en détail à la section 4.1 la sémantique détaillée.

L'intérêt d'un système “multi-threadé”, même sur une machine monoprocesseur, est que l'ordinateur donne l'impression d'effectuer plusieurs tâches en parallèle. Les systèmes d'exploitation modernes (Unix, Windows NT etc.) sont tous multi-threadés donc il y a peu de chances que vous ayez connu l'horreur du monotâche. Le fait de pouvoir ouvrir en même temps `netscape`, `emacs`, et un shell par exemple, et de pouvoir passer d'une fenêtre à l'autre sans problème est la marque d'un tel système. L'application `netscape` même est multi-threadé. Une tâche essaie de se connecter au site choisi, pendant qu'une autre imprime à l'écran etc. Imaginez ce que ce serait si vous ne voyiez rien à l'écran tant que le site auquel vous vous connectez n'a pas fini de vous transmettre toutes les données! En fait, un système d'exploitation comme Unix comporte des processus (lourds) multiples, tels les démons systèmes (ou processus noyau) et souvent un grand nombre de processus (lourds) utilisateurs. Il n'est pas rare d'avoir quelques dizaines voire une centaine de processus lourds sur une machine à tout instant (faire `ps al` par exemple).

Sur une machine multiprocesseurs, des threads peuvent être exécutés sur plusieurs processeurs donc réelement en même temps, quand le système d'exploitation et le support pour les threads sont étudiés pour (c'est le cas pour Windows NT, Solaris 2, Linux etc.). Pour rentrer un peu plus dans les détails, les threads que nous programmerons sont des “threads utilisateurs” qui doivent communiquer avec le noyau de système d'exploitation de temps en temps, ne serait-ce que pour imprimer à l'écran, lire et écrire des fichiers etc. Tout thread utilisateur doit donc être lié d'une façon ou d'une autre à un thread “noyau”. Selon les implémentations, chaque tâche utilisateur peut être liée à une tâche noyau, ou plusieurs tâches utilisateur à plusieurs tâches noyaux, ou encore plusieurs tâches utilisateur à une tâche noyau.

```
class Compte extends Thread {
   int valeur;

   Compte(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
          }
      } catch (InterruptedException e) {
           return;
         }
   }

   public static void main(String[] args) {
      new Compte(1).start();
      new Compte(2000).start();
   }
}
```

FIG. 3.1 –

La première version de Solaris ("green threads") implémentait seulement la dernière possibilité qui est la seule qui ne permet pas de bénéficier de vrai parallélisme sur une architecture multiprocesseur. A partir de Java 1.1 et pour les versions plus récentes de Solaris et de Linux, on est dans le deuxième cas, qui offre le plus de flexibilité et de performances.

## 3.2 Les threads en JAVA

Les threads (ou processus légers) sont définis dans le langage JAVA, et ne sont pas comme en C ou C++, une extension que l'on peut trouver dans différentes bibliothèques.

### 3.2.1 Création

Pour créer un thread, on crée une instance de la classe `Thread`,

`Thread Proc = new Thread();`

Une fois créé, on peut configurer `Proc`, par exemple lui associer une priorité (on en parlera plus à la section 3.3.1). On pourra ensuite l'exécuter en invoquant sa méthode `start`. `start` va à son tour invoquer la méthode `run` du thread. Comme la méthode `run` de la classe `Thread` ne fait rien, il faut la redéfinir. C'est possible par exemple si on définit `Proc` comme une instance d'une sous-classe de `Thread`, dans laquelle on redéfinit la méthode `run`.

En général on a envie qu'un processus contienne des données locales, donc il est vraiment naturel de définir un processus comme une instance d'une sous-classe de `Thread` en général, voir par exemple la figure 3.1

La classe `Compte` gère uniquement ici un entier représentant une somme d'un compte courant d'une banque. Après avoir défini le constructeur, on a écrit une méthode `run` qui sera exécutée deux fois par deux instances différentes de la classe à partir du `main`, l'une avec une valeur de compte initiale égale à `1` et l'autre, à `2000`. `run` se contente d'afficher la valeur courante du compte tous les dixièmes de seconde, jusqu'à une interruption clavier.

L'exécution du programme `main` donne quelque chose comme,

```
> java Compte
1 2000 2000 1 1 1 1
^C
>
```

Néanmoins, cette méthode n'est pas toujours acceptable. En effet dans certains cas, on veut étendre une classe existante et en faire une sous-classe de `Thread` également, pour pouvoir exécuter ses instances en parallèle. Le problème est qu'il n'y a pas d'héritage multiple en JAVA (c'est d'ailleurs plutôt un bien, car c'est très difficile à contrôler, voir C++). Donc on ne peut pas utiliser la méthode précédente. A ce moment là on utilise l'interface `Runnable`.

L'interface `Runnable` représente du code exécutable, et ne possède qu'une méthode,

```
public void run();
```

Par exemple la classe `Thread` implémente l'interface `Runnable`. Mieux encore, on peut construire une instance de `Thread` à partir d'un objet qui implémente l'interface `Runnable` par le constructeur:

```
public Thread(Runnable target);
```

On peut écrire à nouveau la classe `Compte` comme à la figure 3.2

### 3.2.2 Partage des variables

Sur l'exemple de la section précédente il faut noter plusieurs choses. Les entiers `valeur` sont distincts dans les deux threads qui s'exécutent. C'est une variable locale au thread. Si on avait déclaré `static int valeur`, cette variable aurait été partagée par ces deux threads En fait, il aurait été même préférable de déclarer `static volatile int valeur` afin de ne pas avoir des optimisations génantes de la part de `javac`... Si on avait déclaré `Integer valeur` ("classe enveloppante" de `int`), comme à la figure 3.3, cette classe, utilisée en lieu et place de `int` aurait pu être partagée par tous les threads.

### 3.2.3 Quelques fonctions élémentaires sur les threads

Pour s'amuser (disons que c'est au moins utile pour débugger) on peut nommer les threads: `void setName(String name)` nomme le thread courant. `String getName()` renvoie le nom du thread.

Un peu plus utile, on peut recueillir un certain nombre d'informations sur les threads présents à l'exécution. Par exemple, `static Thread currentThread()` renvoie la référence au Thread courant c'est-à-dire celui qui exécute `currentThread()`. `static int enumerate(Thread[] threadArray)` place tous les threads existant (y compris le `main()` mais pas le thread ramasse-miettes) dans le tableau `threadArray` et renvoie leur nombre. `static int activeCount()` renvoie tout simplement le nombre de threads.

```
class Compte implements Runnable {
   int valeur;

   Compte(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
         }
      } catch (InterruptedException e) {
           return;
        }
   }

   public static void main(String[] args) {
      Runnable compte1 = new Compte(1);
      Runnable compte2 = new Compte(2000);
      new Thread(compte1).start();
      new Thread(compte2).start();
   }
}
```

Fig. 3.2 –

```
public class UnEntier {
    int val;

    public UnEntier(int x) {
        val = x;
    }

    public int intValue() {
        return val;
    }

    public void setValue(int x) {
        val = x;
    }
}
```

Fig. 3.3 –

```
class Compte3 extends Thread {
   int valeur;

   Compte3(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
         }
      } catch (InterruptedException e) {
           return; } }

   public static void printThreads() {
      Thread[] ta = new Thread[Thread.activeCount()];
      int n = Thread.enumerate(ta);
      for (int i=0; i<n; i++) {
         System.out.println("Le thread "+ i + " est
                             " + ta[i].getName());
      } }

   public static void main(String[] args) {
      new Compte3(1).start();
      new Compte3(2000).start();
      printThreads(); } }
```

Fig. 3.4 –

Voici un petit exemple d'utilisation (que l'on pourra également trouver, comme tous les autres exemples de ce cours, sur la page web du cours), à la figure 3.4

Son exécution donne:

```
% java Compte3
1 2000 Le thread 0 est main
Le thread 1 est Thread-2
Le thread 2 est Thread-3
1 2000 1 2000 1 2000 1 2000 2000
1 1 2000 2000 1 1 2000 2000 1 1
2000 2000 1 1 2000 2000 1 1 2000
2000 1 1 2000 2000 1 1 ^C
%
```

## 3.3 Eléments avancés

Pour commencer cette section, il faut d'abord expliquer quelles peuvent être les différents états des threads JAVA. Un thread peut être dans l'un des 4 cas suivants:

- l'état initial, comprenant tous les instants entre sa création (par un constructeur) et l'invocation de sa méthode `start()`.
- l'état prêt, immédiatement après que la méthode `start` ait été appelée.
- l'état bloqué représentent les instants où le thread est en attente, par exemple d'un verrou, d'un socket, et également quand il est suspendu (par la méthode `sleep(long)` qui suspend l'exécution du thread pendant un certain temps exprimé en nanosecondes).
- l'état terminaison, quand sa méthode `run()` se termine ou quand on invoque la méthode `stop()` (à éviter si possible, d'ailleurs cette méthode n'existe plus en JAVA 2).

On peut déterminer l'état d'un thread en appelant sa méthode `isAlive()`, qui renvoie un booléen indiquant si un thread est encore vivant, c'est à dire s'il est prêt ou bloqué

On peut aussi interrompre l'exécution d'un thread qui est prêt (passant ainsi dans l'état bloqué). `void interrupt()` envoie une interruption au thread spécifié; si l'interruption se produit dans une méthode `sleep`, `wait` ou `join` (que l'on va voir plus tard), elles lèvent une exception `InterruptedException`[1]; sinon le thread cible doit utiliser une des méthodes suivantes pour savoir si il a été interrompu: `static boolean interrupted()` renvoie un booléen disant si le thread courant a été "interrompu" par un autre ou pas et l'interrompt, `boolean isInterrupted()` renvoie un booléen disant si le thread spécifié a été interrompu ou pas (sans rien faire d'autre).

`void join()` (comme sous Unix) attend la terminaison d'un thread. Egalement: `void join(long timeout)` attend au maximum `timeout` millisecondes. Voir par exemple la figure 3.5

### 3.3.1 Priorités

On peut affecter à chaque processus une priorité qui est un nombre entier, qui plus il est grand, plus le processus est prioritaire. `void setPriority(int priority)` assigne une priorité au thread donné et `int getPriority()` renvoie la priorité d'un thread donné.

1. C'est pourquoi il faut toujours encapsuler ces appels dans un bloc `try ... catch(InterruptedException e)`

```
public class ... extends Thread {
   ...
   public void stop() {
      t.shouldRun=false;
      try {
         t.join();
      } catch (InterruptedException e) {}   } }
```

FIG. 3.5 –

L'idée est que plus un processus est prioritaire, plus l'ordonnanceur JAVA doit lui permettre de s'exécuter tôt et vite. La priorité peut être maximale: `Thread.MAX_PRIORITY`, normale (par défaut): `Thread.NORM_PRIORITY` (au minimum elle vaut 0). Pour bien comprendre cela, il nous faut décrire un peu en détail la façon dont sont ordonnancés les threads JAVA. C'est l'objet de la section suivante.

On peut également déclarer un processus comme étant un démon ou pas:

```
setDaemon(Boolean on);
boolean isDaemon();
```

Un processus démon est typiquement un processus "support" aux autres (par exemple, l'horloge, ou le ramasse-miettes). Il a la propriété d'être détruit quand il n'y a plus aucun processus utilisateur (non-démon) restant (en fait il y a un certain nombre de threads systèmes par JVM: ramasse-miettes, graphique).

### 3.3.2 Ordonnancement des tâches JAVA

Le choix du thread JAVA à exécuter (au moins partiellement) se fait parmi les threads qui sont prêts. Supposons que nous soyons dans le cas monoprocesseur. Il y aura donc à tout moment un seul processus actif à choisir. L'ordonnanceur JAVA est un ordonnanceur préemptif basé sur la priorité des processus. Essayons d'expliquer ce que cela veut dire. "Basé sur la priorité" veut dire qu'à tout moment, l'ordonnanceur essaie de rendre actif le (si on se place d'abord dans le cas simple où il n'y a pas 2 threads de même priorité) thread prêt de plus haute priorité. "Préemptif" veut dire que l'ordonnanceur use de son droit de préemption pour interrompre le thread courant de priorité moindre, qui reste néanmoins prêt. Il va de soit qu'un thread actif qui devient bloqué, ou qui termine rend la main à un autre thread, actif, même s'il est de priorité moindre.

Attention tout de même, un système de priorité n'est en aucun cas une garantie: c'est pourquoi on insiste sur le <u>essaie</u> dans la phrase "l'ordonnanceur essaie de rendre actif le thread prêt de plus haute priorité". On verra au chapitre 5 des algorithmes permettant d'implémenter l'exclusion mutuelle à partir de quelques hypothèses de bases, elles ne font aucunement appel aux priorités qui ne peut pas garantir de telles choses.

Il y a maintenant plusieurs façons d'ordonnancer des threads de même priorité. La spécification ne définit pas cela précisément.

L'ordonnancement "round-robin" (ou "tourniquet") dans lequel un compteur interne fait alterner l'un après l'autre (pendant des périodes de temps prédéfinies) les processus prêts de même priorité. Cela assure l'*équité* dans l'exécution de tous les processus, c'est à dire que tous vont progresser de conserve, et qu'aucun ne sera en *famine* (les processus "plus rapides" étant toujours ordonnancés).

Un ordonnancement plus classique (mais pas équitable en général) est celui où un thread d'une certaine priorité, actif, ne peut pas être prémpté par un thread prêt de même priorité. Il faut que celui-ci passe en mode bloqué pour qu'un autre puisse prendre la main. Cela peut se faire en interrompant régulièrement les processus par des `sleep()` mais ce n'est pas très efficace. Il vaut mieux utiliser `static void yield()`: le thread courant rend la main, ce qui permet à la machine virtuelle JAVA de rendre actif un autre thread de même priorité (on l'utilisera abondament au chapitre 5).

Tout n'est pas complètement aussi simple que cela. L'ordonnancement dépend en fait aussi bien de l'implémentation de la JVM que du système d'exploitation sous-jacent. Il y a deux modèles principaux concernant la JVM. Dans le modèle "green thread", c'est la JVM qui implémente l'ordonnancement des threads qui lui sont associés. Dans le modèle "threads natifs", c'est le système d'exploitation hôte de la JVM qui effectue l'ordonnancement des threads JAVA.

Le modèle green thread est le plus souvent utilisé dans les systèmes UNIX donc même sur une machine multi-processeur, seul un thread JAVA sera exécuté à la fois sur un tel système.

Par contre sur les plateformes Windows (Windows 95 ou Windows NT), les threads sont des threads natifs, et les threads JAVA correspondent bijectivement à des threads noyaux.

Sous Solaris, la situation est plus complexe, car il y a un niveau intermédiaire de processus connu de système d'exploitation (les "light-weight processes"). On n'entrera pas ici dans les détails.

Pour ceux qui sont férus de système, disons que l'on peut contrôler pas mal de choses de ce côté là grâce à la Java Native Interface, voir par exemple

`http://java.sun.com/docs/books/tutorial/native1.1/`

### 3.3.3 Les groupes de processus

JAVA permet de définir des groupes de processus. Il existe une classe `class ThreadGroup` qui représente précisément ces groupes de processus. On peut définir un nouveau Thread dans un groupe donné par les constructeurs,

```
new Thread(ThreadGroup group, Runnable target);
new Thread(ThreadGroup group, String name);
new Thread(ThreadGroup group, Runnable target, String name);
```

Par défaut, un thread est créé dans le groupe courant (c'est à dire de celui qui l'a créé). Au démarrage, un thread est créé dans le groupe de nom `main`. On peut créer de nouveaux groupes et créer des threads appartenant à ces groupes par de nouveaux constructeurs. Par exemple,

```
ThreadGroup Groupe = new ThreadGroup("Mon groupe");
Thread Processus = new Thread(Groupe, "Un processus");
...
ThreadGroup MonGroupe = Processus.getThreadGroup();
```

On peut également faire des sous-groupes (en fait n'importe quelle arborescence est autorisée). On peut alors demander un certain nombre d'informations:

```
Groupe.getParent();
Groupe.getName();
Groupe.activeCount();
Groupe.activeGroupCount();
Groupe.enumerate(Thread[] list);
Groupe.enumerate(Thread[] list, boolean recurse);
Groupe.enumerate(ThreadGroup[] list);
Groupe.enumerate(ThreadGroup[] list, boolean recurse);
```

La première instruction permet de connaître le groupe parent, la deuxième le nom du groupe (une `String`), la troisième le nombre de thread du groupe `Groupe`, la quatrième, le nombre de groupe de thread, et les quatre suivantes, d'énumérer soit les threads d'un groupe, soit les groupes (cela remplit le tableau correspondant à chaque fois). Le booléen `recurse` indique si on désire avoir la liste récursivement, pour chaque sous-groupe etc.

# Chapitre 4

# Coordination de processus

## 4.1 Problème

Modifions notre classe `Compte` afin de pouvoir faire des opérations dessus, voir figure 4.1.

Une exécution typique est la suivante:

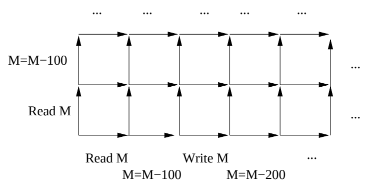
```
% java Banque
Conseiller Mari: Voici vos 100 francs.
Conseiller Femme: Voici vos 100 francs.
Conseiller Mari: Il vous reste 800 francs.
Conseiller Femme: Il vous reste 800 francs.
Conseiller Mari: Voici vos 200 francs.
Conseiller Femme: Voici vos 200 francs.
Conseiller Femme: Il vous reste 400 francs.
Conseiller Mari: Il vous reste 400 francs.
Conseiller Mari: Voici vos 300 francs.
Conseiller Femme: Voici vos 300 francs.
Conseiller Femme: Vous etes fauches!
Conseiller Mari: Vous etes fauches! ...
```

Conclusion: le mari a retiré 600 francs du compte commun, la femme a retiré 600 francs du compte commun, qui ne contenait que 1000 francs au départ! La banque ne va pas être contente...

L'explication est simple et nous entraîne à faire un peu de sémantique.

L'exécution de plusieurs threads se fait en exécutant une action insécable ("atomique") de l'un des threads, puis d'un autre ou d'éventuellement du même etc. Tous les "mélanges" possibles sont permis

C'est ce que l'on appelle la sémantique par entrelacements; on en représente une portion ci-dessous:




Donnons une idée de la façon de donner une sémantique à un fragment du langage JAVA (et en supposant les lectures et écritures atomiques). Nous allons prendre comme langage un langage simple qui comprend comme constructions (étant donné

```
public class Banque implements Runnable {
    Compte nom;

    Banque(Compte n) {
         nom = n; }

    public void Liquide (int montant)
           throws InterruptedException {
         if (nom.retirer(montant)) {
             Thread.currentThread().sleep(50);
             Donne(montant);
             Thread.currentThread().sleep(50); }
         ImprimeRecu();
         Thread.currentThread().sleep(50); }

    public void Donne(int montant) {
         System.out.println(Thread.currentThread().
           getName()+": Voici vos " + montant + " francs."); }

    public void ImprimeRecu() {
         if (nom.solde() > 0)
            System.out.println(Thread.currentThread().
              getName()+": Il vous reste " + nom.solde() + " francs.");
         else
            System.out.println(Thread.currentThread().
              getName()+": Vous etes fauches!");
    }

    public void run() {
        try {
           for (int i=1;i<10;i++) {
                Liquide(100*i);
                Thread.currentThread().sleep(100+10*i);
           }
        } catch (InterruptedException e) {
            return;
        }
    }

    public static void main(String[] args) {
        Compte Commun = new Compte(1000);
        Runnable Mari = new Banque(Commun);
        Runnable Femme = new Banque(Commun);
        Thread tMari = new Thread(Mari);
        Thread tFemme = new Thread(Femme);
        tMari.setName("Conseiller Mari");
        tFemme.setName("Conseiller Femme");
        tMari.start();
        tFemme.start();
    }
}
```

FIG. 4.1 –

une syntaxe pour les **CONSTANTE**s entières et un ensemble fini de **VARIABLE**s également entières):

| | | | | |
|---|---|---|---|---|
| EXPR | ::= | CONSTANTE | \| | VARIABLE |
| | \| | EXPR+EXPR | \| | EXPR*EXPR |
| | \| | EXPR/EXPR | \| | EXPR-EXPR |
| TEST | ::= | EXPR==EXPR | \| | EXPR < EXPR |
| | \| | EXPR > EXPR | \| | TEST && TEST |
| | \| | TEST \| \| TEST | \| | ! TEST |
| INSTR | ::= | VARIABLE=EXPR | | |
| BLOCK | ::= | $\epsilon$ | \| | INSTR;BLOCK |
| | \| | if TEST<br>then BLOCK<br>else BLOCK ;BLOCK | | |
| | \| | while TEST<br>BLOCK ;<br>BLOCK | | |

Appelons $L$ le langage engendré par la grammaire plus haut. On définit la sémantique d'un **BLOCK** inductivement sur la syntaxe, en lui associant ce que l'on appelle un système de transitions. Un système de transitions est un 4-uplet $(S, i, E, Tran)$ où,

- $S$ est un ensemble d'états,
- $i \in S$ est l'"état initial",
- $E$ est un ensemble d'étiquettes,
- $Tran \subseteq S \times E \times S$ est l'ensemble des transitions. On note plus communément $(s, t, s') \in Tran$ comme $s \xrightarrow{t} s'$.

En fait, on représente généralement un système de transitions comme un graphe orienté et étiqueté, comme le suggère la représentation graphique de la relation de transition $Tran$, que nous avons déjà vue.

Donnons maintenant les règles de formation du système de transition correspondant à un **BLOCK**. Tout d'abord appelons environnement toute fonction de **VARIABLE** vers l'ensemble des entiers relatifs. Un état du système de transition que nous allons construire sera une paire $(l, \rho)$ où $l \in L$ et $\rho$ est un environnement. Intuitivement, $l$ est le programme qu'il reste à exécuter et $\rho$ est l'état courant de la machine, c'est à dire la valeur courante de toutes ses variables. On a alors:

-
$$(x = expr; block, \rho) \xrightarrow{x = expr} (block, \rho[x \to [\![expr]\!](\rho)])$$
où $[\![expr]\!](\rho)$ dénote la valeur entière résultat de l'évaluation de l'expression $expr$ dans l'environnement $\rho$ (on ne donne pas les règles formelles évidentes qui donnent un sens précis à cela...).
-
$$(if\ t\ then\ b_1\ else\ b_2; b_3, \rho) \xrightarrow{then} (b_1; b_3, \rho)$$
si $[\![t]\!]\rho = vrai$ sinon
$$(if\ t\ then\ b_1\ else\ b_2; b_3, \rho) \xrightarrow{else} (b_2; b_3, \rho)$$
-
$$(while\ t\ b_1;\ b_2, \rho) \xrightarrow{while} (b_1; while\ t\ b_1;\ b_2, \rho)$$
si $[\![t]\!]\rho = vrai$ sinon
$$(while\ t\ b_1;\ b_2, \rho) \xrightarrow{while} (b_2, \rho)$$

Maintenant, supposons que l'on ait un ensemble de processus $P_1, \ldots, P_n$ qui chacun sont écrits dans le langage $L$. On peut encore donner une sémantique opérationnelle (en termes de systèmes de transition) dite par "entrelacements" en posant:

- A chaque $P_j$ on associe par $[\![.]\!]$ défini plus haut un système de transition $(S_j, i_j, E_j, Tran_j)$,
- On définit $(S, i, E, Tran) = [\![P_1 \mid \ldots \mid P_n]\!]$ par:
  - $S = S_1 \times \ldots \times S_n$,
  - $i = (i_1, \ldots, i_n)$,
  - $E = E_1 \cup \ldots \cup E_n$,
  -

$$Tran = \left\{ \begin{array}{c} (s_1, \ldots, s_j, \ldots, s_n) \xrightarrow{t_j} (s_1, \ldots, s'_j, \ldots, s_n) \\ \text{où } s_j \xrightarrow{\quad t_j \quad} s'_j \end{array} \in Tran_j \right\}$$

Les exécutions ou traces ou entrelacement dans cette sémantique sont des suites finies:

$$s_0 = i \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \quad \ldots \xrightarrow{t_n} s_n$$

telles que $s_{i-1} \xrightarrow{t_i} s_i \in Tran$ pour tout $i$ entre 1 et $n$.

On voit dans un des entrelacements que si les 2 threads `tMari` et `tFemme` sont exécutés de telle façon que dans `retirer`, chaque étape soit faite en même temps, alors le test pourra être satisfait par les deux threads en même temps, qui donc retireront en même temps de l'argent.

## 4.2 Une solution: `synchronized`

Une solution est de rendre "atomique" le fait de `retirer` de l'argent, Cela se fait en JAVA en déclarant "synchronisée" la méthode `retirer` de la classe `Compte`:

```
public synchronized boolean retirer(int somme)
```

Maintenant l'exécution typique paraît meilleure:

```
% java Banque
Conseiller Mari: Voici vos 100 francs.
Conseiller Mari: Il vous reste 800 francs.
Conseiller Femme: Voici vos 100 francs.
Conseiller Femme: Il vous reste 800 francs.
Conseiller Mari: Voici vos 200 francs.
Conseiller Mari: Il vous reste 400 francs.
Conseiller Femme: Voici vos 200 francs.
Conseiller Femme: Il vous reste 400 francs.
Conseiller Femme: Il vous reste 100 francs.
Conseiller Mari: Voici vos 300 francs.
Conseiller Mari: Il vous reste 100 francs.
Conseiller Femme: Il vous reste 100 francs.
Conseiller Mari: Il vous reste 100 francs...
```

Le résultat est maintenant que le mari a tiré 600 francs, la femme a tiré 300 francs, et il reste bien 100 francs dans le compte commun, ouf!

Remarquez que `synchronized` peut aussi s'utiliser sur une morceau de code: étant donne un objet `obj`:

```
synchronized(obj) {
   BLOCK }
```

```
public class Semaphore {
    int n;
    String name;

    public Semaphore(int max, String S) {
        n = max;
        name = S;
    }

    public synchronized void P() {
        if (n == 0) {
            try {
                 wait();
            } catch(InterruptedException ex) {};
        }
        n--;
        System.out.println("P("+name+")");
    }

    public synchronized void V() {
        n++;
        System.out.println("V("+name+")");
        notify();
    }
}
```

FIG. 4.2 –

permet d'exécuter `BLOCK` en exclusion mutuelle.

## 4.3 Moniteurs

Chaque objet fournit un verrou, mais aussi un mécanisme de mise en attente (forme primitive de communication inter-threads; similaire aux variables de conditions ou aux moniteurs [Hoa74]).

`void wait()` attend l'arrivée d'une condition sur l'objet sur lequel il s'applique (en général `this`). Il doit être appelé depuis l'intérieur d'une méthode ou d'un bloc `synchronized` (il y a aussi une version avec timeout). `void notify()` notifie un thread en attente d'une condition, de l'arrivée de celle-ci. De même, cette méthode doit être appelée depuis une méthode déclarée `synchronized`. Enfin, `void notify_all()` fait la même chose mais pour tous les threads en attente sur l'objet.

## 4.4 Sémaphores

### 4.4.1 Sémaphores binaires

Cela permet de réaliser l'exclusion mutuelle, donc l'accès par des processus à des *sections critiques*, comme on le montre à la figure 4.3.

Une exécution typique de ce programme (sans P, V) est la suivante:

```
% java essaiPV
Thread-2: x=4
```

```
public class essaiPV extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPV(Semaphore s) {
        u = s;
    }

    public void run() {
        int y;
        // u.P();

        try {
            Thread.currentThread().sleep(100);
            y = x;
            Thread.currentThread().sleep(100);
            y = y+1;
            Thread.currentThread().sleep(100);
            x = y;
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
        System.out.println(Thread.currentThread().
                           getName()+": x="+x);
        // u.V();
    }

    public static void main(String[] args) {
        Semaphore X = new Semaphore(1,"X");
        new essaiPV(X).start();
        new essaiPV(X).start();
    }
}
```

FIG. 4.3 –

```
Thread-3: x=4
```

Maintenant, avec P, V on a par exemple:

```
% java essaiPV
P(X)
Thread-2: x=4
V(X)
P(X)
Thread-3: x=5
V(X)
```

### 4.4.2 Un peu de sémantique

On va rajouter à notre langage $L$ les primitives $P(.)$ et $V(.)$. On suppose que la syntaxe pour chacun des processus est bien formée dans le sens suivant:

–

–

La sémantique par systèmes de transitions change assez peu:

- On rajoute dans l'ensemble d'états $S$ du système de transitions une composante nouvelle qui est une fonction $\kappa : VARIABLE \rightarrow \{0,1\}$. Donc $S = (S_1, \ldots, S_n, \kappa)$. Cette fonction $\kappa$ represente le fait de savoir si chaque variable est verrouillée par un processus ou non,
- $i = (P_1, \ldots, P_n, \kappa_0)$ où $\kappa_0$ est la fonction qui donne uniformément 1,
- $E$ contient maintenant les nouvelles actions $Px$ et $Vx$ pour toutes les variables $x$ du programme,
- On a de nouvelles transitions dans $Tran$ de la forme:

$$(b_1, \ldots, Px; b_j, \ldots, b_n, \kappa) \xrightarrow{Px} (b_1, \ldots, b_j, \ldots, b_n, \kappa')$$

  si $\kappa(x) = 1$; et alors $\kappa'(y) = \kappa(x)$ pour tout $y \neq x$ et $\kappa'(x) = 0$, et,

$$(b_1, \ldots, Px; b_j, \ldots, b_n, \kappa) \xrightarrow{Vx} (b_1, \ldots, b_j, \ldots, b_n, \kappa')$$

  avec $\kappa'(y) = \kappa(x)$ pour tout $y \neq x$ et $\kappa'(x) = 1$.

L'apparition de ces nouvelles transitions contraint en fait les entrelacements possibles. Si on "abstrait" ce système de transition (et le langage de base) à de simples suites de verrouillages et déverrouillages de variables partagées, on peut facilement se rendre compte de ce qui ce qui peut se passer.

Un des problèmes les plus courants est d'avoir voulu trop protéger les accès partagés et à aboutir à une situation où les processus se bloquent mutuellement; on dit alors qu'il y a une étreinte fatale, ou encore que le système de transition a des points morts (des états desquels aucune transition n'est possible, mais qui ne sont pas des états finaux autorisés: ces états finaux autorisées seraient uniquement en général des $n$-uplets des états finaux de chacun des processus).

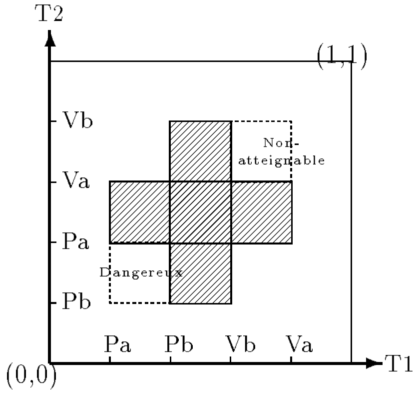
Un exemple classique est donné par le programme "abstrait" suivant constitué des deux processus:

```
A=Pa;Pb;Va;Vb
B=Pb;Va;Vb;Va
```

En regardant la sémantique par entrelacements, on trouve assez vite qu'une exécution synchrone de `A` et de `B` rencontre un point mort: en effet dans ce cas `A` (respectivement `B`) acquiert `a` (respectivement `b`) et attend ensuite la resource `b` (respectivement `a`).

Une image pratique (qui s'abstrait élégamment de la lourdeur de la représentation des entrelacements) est d'utiliser une image géométrique continue, les "progress graphs" (E.W.Dijkstra (1968), ou encore les "graphes d'avancement".

Associons en effet à chaque processus une coordonnée dans un espace $\mathbb{R}^n$ ($n$ étant le nombre de processus). Chaque coordonnée $x_i$ représente en quelque sorte le temps local du $i$ème processus. Les états du système parallèle sont donc des point dans cet espace $\mathbb{R}^n$ (inclus dans un hyper-rectangle dans les bornes en chaque coordonnées sont les bornes des temps locaux d'exécution de chaque processus). Les traces d'exécution sont les chemins continus et croissants dans chaque coordonnée. Dans la figure suivante, on a représenté le graphe de processus correspondant à nos deux processus `A` et `B`.




On constate que certains points dans le carré unité ne sont pas des états valides du programme parallèle. Prenons un point qui a pour coordonnée $x_1$ un temps local compris entre $P_b$ et $V_b$ et pour coordonnée $x_2$ un temps local compris également entre $P_b$ et $V_b$ (mais cette fois pour le processus `B`). Il ne peut être valide car dans cet état, les deux processus auraient chacun un verrou sur le même objet `b`. Cela contredit la sémantique par système de transitions vue auparavant, c'est à dire que cela contredit la propriété d'exclusion mutuelle sur `b`. On doit également interdire le rectangle horizontal, à cause de la propriété d'exclusion mutuelle sur la variable partagée `a`.

On s'aperçoit maintenant que tout chemin d'exécution (partant donc du point initial en bas à gauche du graphe d'avancement) et qui rentrerait dans le carré marqué "dangereux" doit nécessairement (par sa propriété de croissante en chaque coordonnée et par sa continuité) aboutir au point mort, minimum des intersections des deux rectangles interdits. Cette concavité inférieure est responsable de l'étreinte fatale entre les processus `A` et `B`. De même on s'aperçoit que certains états, dans la concavité supérieure de la région interdite, sont des états inatteignables depuis l'état initial. On reviendra la-dessus et sur des critères plus simples pour trouver si un système a des points morts à la section 4.4.4. Sachez simplement que ces idées apparemment simples peuvent être développées extrêmement loin!

Remarquez également que `synchronized` de JAVA ne fait jamais que `P` du verrou associé à une méthode ou à un block de code, puis `V` de ce même verrou à la fin de ce même code. Une méthode `M` de la classe `A` qui est `synchronized` est équivalente à `P(A)`, corps de `M`, `V(A)`. Il y a malgré tout une subtile différence. La construction `synchronized` est purement déclarative (contrairement à notre construction de `P` et à donc une portée définie statiquement, contrairement à l'utilisation de `P` et `V`.

### 4.4.3 Un complément sur la JVM

Pour bien comprendre la sémantique des communications inter-threads en JAVA, il nous faut expliquer un peu en détail comment la JVM traite les accès mémoire des threads.

Toutes les variables (avec leurs verrous correspondants et la liste des threads en attente d'accès) de tous les threads sont stockées dans la mémoire gérée par la JVM correspondante. Bien évidemment, les variables locales d'un thread ne peuvent pas être accédée par d'autres threads, donc on peut imaginer que chaque thread a une mémoire locale et a accès à une mémoire globale.

Pour des raisons d'efficacité, chaque thread a une copie locale des variables qu'il doit utiliser, et il faut bien sûr que des mises à jour de la mémoire principale (celle gérée par la JVM) soit effectuées, de façon cohérente.

Plusieurs actions *atomiques* (c'est à dire non interruptibles, ou encore qui ne s'exécuteront qu'un à la fois sur une JVM) peuvent être effectuées pour gérer ces copies locales et la mémoire principales, au moins pour les types simples (`int`, `float`...). Ces actions décomposent les accès mémoires qui sont nécessaires à l'interprétation d'une affectation JAVA:

- les actions qu'un thread peut exécuter sont:
  - *use* transfère le contenu de la copie locale d'une variable à la machine exécutant le thread.
  - *assign* transfère une valeur de la machine exécutant le thread à la copie locale d'une variable de ce même thread.
  - *load* affecte une valeur venant de la mémoire principale (après un *read*, voir plus loin) à la copie locale à un thread d'une variable.
  - *store* transfère le contenu de la copie locale à un thread d'une variable à la mémoire principale (qui va ainsi pouvoir faire un *write*, voir plus loin).
  - *lock* réclame un verrou associé à une variable à la mémoire principale.
  - *unlock* libère un verrou associé à une variable.
- les actions que la mémoire principale (la JVM) peuvent exécuter sont:
  - *read* transfère le contenu d'une variable de la mémoire principale vers la mémoire locale d'un thread (qui pourra ensuite faire un `load`).
  - *write* affecte une valeur transmise par la mémoire locale d'un thread (par *store*) d'une variable dans la mémoire principale.

Pour les types "plus longs" comme `double` et `long`, la spécification permet tout à fait que *load*, *store*, *read* et *write* soient non-atomiques. Elles peuvent par exemple être composées de deux *load* atomiques de deux "sous-variables" *int* de 32 bits. Le programmeur doit donc faire attention à synchroniser ses accès à de tels types, s'ils sont partagés par plusieurs threads.

L'utilisation par la JVM de ces instructions élémentaires est soumise à un certain nombre de contraintes, qui permettent de définir la sémantique des accès aux ressources partagées de JAVA. Tout d'abord, pour un thread donné, à tout *lock* correspond plus tard une opération *unlock*. Ensuite, chaque opération *load* est associée à une opération *read* qui la précède; de même chaque *store* est associé à une opération *write* qui la suit.

Pour une variable donnée et un thread donné, l'ordre dans lequel des *read* sont fait par la mémoire locale (respectivement les *write*) est le même que l'ordre dans lequel le thread fait les *load* (respectivement les *store*). Attention, ceci n'est vrai que pour une variable donnée! Pour les variables *volatile*, les règles sont plus contraignantes car l'ordre est aussi respecté entre les *read*/*load* et les *write*/store entre les variables. Pour les *volatile* on impose également que les *use* suivent immédiatement les *load* et que les `store` suivent immédiatement les *assign*.

```
class Essai1 extends Thread {
  public Essai1(Exemple1 e) {
    E = e;
  }

  public void run() {
    E.F1();
  }

  private Exemple1 E;
}
class Essai2 extends Thread {
  public Essai2(Exemple1 e) {
    E = e;
  }

  public void run() {
    E.G1();
  }

  private Exemple1 E;
}

class Lance {
  public static void main(String args[]) {
    Exemple1 e = new Exemple1();

    Essai1 T1 = new Essai1(e);
    Essai2 T2 = new Essai2(e);

    T1.start();
    T2.start();
    System.out.println("From Lance a="+e.a+" b="+e.b);
  }
}
```

FIG. 4.4 –

Pour bien comprendre tout ça prenons deux exemples classiques, aux figures 4.5 et 4.6 respectivement. On utilise ces classes dans deux threads de la même façon, on en donne donc le code à la figure 4.4 que dans le cas de `Exemple1`.

Une exécution typique est:

```
> java Lance
a=2 b=2
```

Mais il se peut que l'on trouve également:

```
> java Lance
a=1 b=1
```

ou encore

```
> java Lance
a=2 b=1
```

```
class Exemple1 {
  int a = 1;
  int b = 2;

  void F1() {
    a = b;
  }

  void G1() {
    b = a;
  }
}
```

FIG. 4.5 –

```
class Exemple2 {
  int a = 1;
  int b = 2;

  synchronized void F2() {
    a = b;
  }

  synchronized void G2() {
    b = a;
  }
}
```

FIG. 4.6 –

En fait, pour être plus précis, les contraintes d'ordonnancement font que les chemins possibles d'exécution sont les entrelacements des 2 traces séquentielles

$$read\ b \longrightarrow load\ b \longrightarrow use\ b \longrightarrow assign\ a \longrightarrow store\ a \longrightarrow write\ a$$

pour le thread `Essai1` et

$$read\ a \longrightarrow load\ a \longrightarrow use\ a \longrightarrow assign\ b \longrightarrow store\ b \longrightarrow write\ b$$

pour le thread `Essai2`. Appelons $a1$, $b1$ (respectivement $a2$ et $b2$) les copies locales des variables $a$ et $b$ pour le thread `Essai1` (respectivement pour le thread `Essai2`). Alors on peut avoir les entrelacement suivants (modulo les commutations des *read a* et *read b*, ce qui n'a aucune influence sur le calcul):

$$write\ a \longrightarrow read\ a \longrightarrow read\ b \longrightarrow write\ b$$

$$read\ a \longrightarrow write\ a \longrightarrow write\ b \longrightarrow read\ b$$

$$read\ a \longrightarrow write\ a \longrightarrow read\ b \longrightarrow write\ b$$

Dans le premier cas l'état de la machine est $la1 = 2$, $lb1 = 2$, $a = 2$, $b = 2$, $la2 = 2$ et $lb2 = 2$. Dans le deuxième on a $la1 = 1$, $lb1 = 1$, $a = 1$, $b = 1$, $la2 = 1$ et $lb2 = 1$. Enfin dans le troisième, on a $la1 = 2$, $lb1 = 2$, $a = 2$, $b = 1$, $la1 = 1$ et $lb1 = 1$.

Si on utilise la version synchronisée de `Exemple1`, qui est `Exemple2` à la figure 4.6, alors on n'a plus que les deux premiers ordonnancements qui soient possibles:

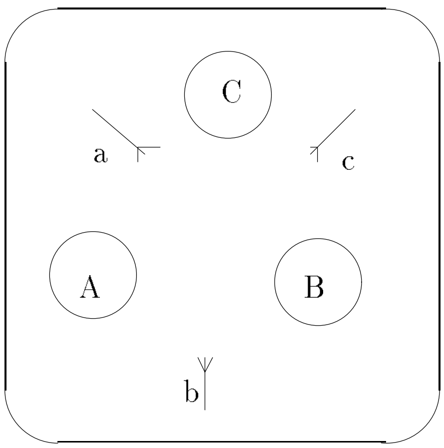
$$write\ a \longrightarrow read\ a \longrightarrow read\ b \longrightarrow write\ b$$

$$read\ a \longrightarrow write\ a \longrightarrow write\ b \longrightarrow read\ b$$

### 4.4.4 Quelques grands classiques

Exerçons un peu nos talents sur des problèmes classiques du parallélisme (que vous aurez peut-être vu par ailleurs, par exemple en cours de système d'exploitation).

Le premier problème est celui des philosophes qui dînent (introduit par E. W. Dijkstra [Dij68]). Dans le cas général, on a $n$ philosophes assis autour d'une table ronde:



avec chacun une fourchette à sa gauche et une autre à sa droite. On n'a pas représenté ici le sujet principal: le bol de spaghettis supposé être au centre. Les philosophes voulant manger, doivent utiliser les 2 fourchettes en même temps pour se servir de spaghettis. Un des premiers "protocoles" venant à l'esprit est celui dans lequel chaque philosophe essaie d'attraper sa fourchette gauche, puis droite, se sert et enfin repose la fourchette gauche puis droite. Cela donne en JAVA le code de la figure 4.7.

Une exécution typique est:

```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```

On constate un interblocage des processus. Comment aurions nous pu nous en douter? L'explication est simple: quand les trois philosophes prennent en même temps leur fourchette gauche, ils attendent tous leur fourchettes droites...qui se trouvent être les fourchettes gauches de leur voisin de droite.

On s'aperçoit que ce point mort est simplement dû à un cycle dans les demandes et attributions des ressources partagées. Essayons de formaliser un peu cela.

Construisons un graphe orienté (le "graphe de requêtes") dont les noeuds sont les resources et dont les arcs sont comme suit. On a un arc d'un noeud $a$ vers un noeud $b$ si il existe un processus qui ayant acquis un verrou sur $a$ (sans l'avoir relaché) demande un verrou sur $b$. On étiquette alors cet arc par le nom du processus en question. Alors on peut facilement montrer que si le graphe de requête est *acyclique* alors le système constitué par les threads considérés ne peut pas rentrer en interblocage. En effet, en abstrayant le programme JAVA pour ne s'intéresser plus qu'aux suites de vérrouillages et dévérrouillages, on obtient les trois processus suivants en parallèle:

```
A=Pa;Pb;Va;Vb
B=Pb;Pc;Vb;Vc
C=Pc;Pa;Vc;Va
```

Un exemple en est donné à la figure 4.8, correspondant aux 3 philosophes, et ce graphe est bien cyclique.

On pouvait aussi s'en apercevoir sur le graphe d'avancement: le point mort est du à l'intersection de 3 (=nombre de processus) hyperrectangles, ou encore la concavité "inférieure de la figure 4.10... mais cela nous entraîne trop loin.

La condition sur les graphes de requêtes est une condition suffisante à l'absence de points morts, et non nécessaire comme le montre l'exemple suivant (où `A`, `B` et `C` sont encore 3 threads exécutés en parallèle):

```
A=Px;Py;Pz;Vx;Pw;Vz;Vy;Vw
B=Pu;Pv;Px;Vu;Pz;Vv;Vx;Vz
C=Py;Pw;Vy;Pu;Vw;Pv;Vu;Vv
```

dont on trouve le graphe de requêtes (fortement cyclique!) à la figure 4.11. Pour avoir une idée de pourquoi cette politique d'accès aux ressources partagées n'a pas de point mort, il faut regarder le graphe d'avancement, représenté à la figure 4.12. La région interdite est homéomorphe à un tore (plus ou moins sur l'antidiagonale du cube... c'est une image bien sûr!) et le chemin passant au milieu du tore n'est jamais stoppé par une concavité intérieure.

Comment remédier au problème d'interblocage dans le cas des $n$ philosophes? Il faut réordonnancer les accès aux variables partagées. Il y a beaucoup de solutions

```
public class Phil extends Thread {
    Semaphore LeftFork;
    Semaphore RightFork;

    public Phil(Semaphore l, Semaphore r) {
        LeftFork = l;
        RightFork = r;
    }

    public void run() {
        try {
            Thread.currentThread().sleep(100);
            LeftFork.P();
            Thread.currentThread().sleep(100);
            RightFork.P();
            Thread.currentThread().sleep(100);
            LeftFork.V();
            Thread.currentThread().sleep(100);
            RightFork.V();
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
    }
}

public class Dining {

    public static void main(String[] args) {
        Semaphore a = new Semaphore(1,"a");
        Semaphore b = new Semaphore(1,"b");
        Semaphore c = new Semaphore(1,"c");
        Phil Phil1 = new Phil(a,b);
        Phil Phil2 = new Phil(b,c);
        Phil Phil3 = new Phil(c,a);
        Phil1.setName("Kant");
        Phil2.setName("Heidegger");
        Phil3.setName("Spinoza");
        Phil1.start();
        Phil2.start();
        Phil3.start();
    } }
```

Fig. 4.7 –

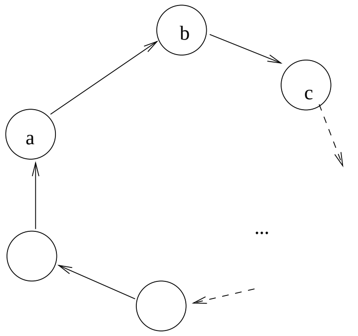



FIG. 4.8 – *Le graphe de requêtes pour les 3 philosophes.*

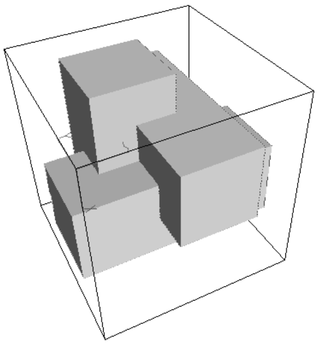


FIG. 4.9 – *Graphe d'avancement des 3 philosophes*

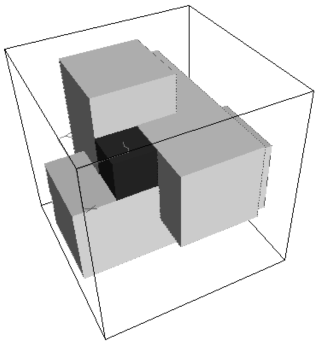


FIG. 4.10 – *Point mort des 3 philosophes*

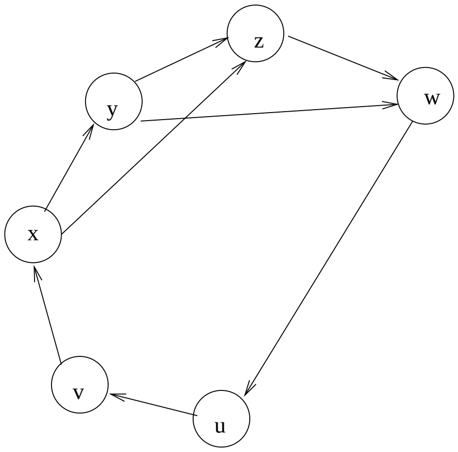



FIG. 4.11 – *Graphe de requêtes pour l'exemple Lipsky/Papadimitriou.*

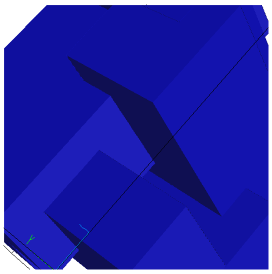


FIG. 4.12 –

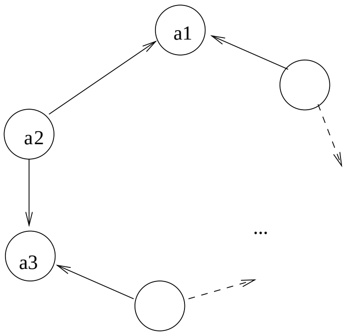



FIG. 4.13 – *Le nouveau graphe de requêtes.*

(éventuellement avec ordonnanceur extérieur). En fait il y en a une très simple: on numérote les philosophes, par exemple dans le sens trigonométrique autour de la table. On impose maintenant que les philosophes de numéro pair acquièrent la fourchette gauche d'abord, puis la droite et aussi que les philosophes de numéro impair acquièrent la fourchette droite d'abord, puis la gauche ensuite. Cela donne (en terme de `P` et de `V`):

```
P1=Pa2;Pa1;Va2;Va1
P2=Pa2;Pa3;Va2;Va3
...
P2k=Pa2k;Pa(2k+1);Va2k;Va(2k+1)
P(2k+1)=Pa(2k+2);Pa(2k+1);Va(2k+2);Va(2k+1)
```

Le graphe de requêtes est décrit à la figure 4.13, on voit bien qu'il est acyclique, donc que le système de processus ne rencontre pas de points morts.

### 4.4.5 Sémaphores à compteur

On a parfois besoin d'objets qui peuvent être partagés par $n$ processus en même temps et pas par $n+1$. C'est une généralisation évidente des sémaphores binaires (qui sont le cas $n = 1$). Par exemple, on peut avoir à protéger l'accès à des tampons de messages de capacité $n$. On appelle les sémaphores correspondants, les sémaphores à compteur. Il sont en fait implémentables à partir des sémaphores binaires [Dij68], comme le fait le code de la figure 4.14.

Par exemple:

```
% java essaiPVsup
```

```
public class essaiPVsup extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPVsup(Semaphore s) {
        u = s;
    }

    public void run() {
        int y;
        u.P();
        try {
            Thread.currentThread().sleep(100);
            y = x;
            Thread.currentThread().sleep(100);
            y = y+1;
            Thread.currentThread().sleep(100);
            x = y;
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
        System.out.println(Thread.currentThread().
                               getName()+": x="+x);
        u.V(); }

    public static void main(String[] args) {
        Semaphore X = new Semaphore(2,"X");
        new essaiPVsup(X).start();
        new essaiPVsup(X).start();
    }
}
```

FIG. 4.14 –

```
Thread-2: P(X)
Thread-3: P(X)
Thread-2: x=4
Thread-2: V(X)
Thread-3: x=4
Thread-3: V(X)
```

On voit bien que l'on n'a pas d'exclusion mutuelle.

Pour les tampons de capacité bornée cela peut servir comme on l'a déjà dit (problème du type producteur consommateur) et comme on le montre par un petit code JAVA à la figure 4.15.

Dans l'exemple précédent, on a 3 processus qui partagent une file à 2 cellules protégée (en écriture) par un 2-sémaphore. Chaque processus essaie de produire (`push`) et de consommer (`pop`) des données dans cette file. Pour que la file ne perde jamais de messages, il faut qu'elle ne puisse pas être par plus de 2 processus en même temps.

Un exemple typique d'exécution est le suivant:

```
% java essaiPVsup2
Thread-2: P(X)
Thread-2: push
Thread-3: P(X)
Thread-3: push
Thread-2: pop
Thread-3: pop
Thread-2: V(X)
Thread-3: V(X)
Thread-4: P(X)
Thread-4: push
Thread-4: pop
Thread-4: V(X)
```

## 4.5 Barrières de synchronisation

Un autre mécanisme de protection, ou plutôt de synchronisation qui est couramment utilisé dans les applications parallèles est ce qu'on appelle une barrière de synchronisation. Son principe est que tout processus peut créer une barrière de synchronisation (commune à un groupe de processus) qui va permettre de donner un rendez-vous à exactement $n$ processus. Ce point de rendez-vous est déterminé par l'instruction `waitForRest()` dans le code de chacun des processus. On en donne une implémentation et un code (tiré du livre [OW00]) à la figure 4.16.

Un exemple d'utilisation typique est donné à la figure 4.17.

## 4.6 Un exemple d'ordonnanceur: séquentialisation

Considérez le problème suivant. On a une base de données distribuée qui contient les réservations aériennes d'une grande compagnie. Les clients réservent leurs billets au travers d'agences dans le monde entier, qui donc n'ont aucun moyen direct de se synchroniser. Les informations sur les réservations sont dans une mémoire d'ordinateur unique quelque part sur la planète. Supposons que nous ayons deux clients, Bill et Tom, le premier habitant Paris et désirant se rendre à New York (en réservant son billet depuis l'agence de Paris), et le deuxième habitant New York et désirant se rendre à Paris (en réservant son billet depuis l'agence de New York). Supposons

```
public class essaiPVsup2 extends Thread {
    static String[] x = {null,null};
    Semaphore u;

    public essaiPVsup2(Semaphore s) {
        u = s;
    }

    public void push(String s) {
        x[1] = x[0];
        x[0] = s;
    }

    public String pop() {
        String s = x[0];
        x[0] = x[1];
        x[1] = null;
        return s; }

    public void produce() {
        push(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().
                           getName()+": push");
    }

    public void consume() {
        pop();
        System.out.println(Thread.currentThread().
                           getName()+": pop"); }

    public void run() {
        try {
            u.P();
            produce();
            Thread.currentThread().sleep(100);
            consume();
            Thread.currentThread().sleep(100);
            u.V();
        } catch(InterruptedException e) {};
    }

    public static void main(String[] args) {
        Semaphore X = new Semaphore(2,"X");
        new essaiPVsup2(X).start();
        new essaiPVsup2(X).start();
        new essaiPVsup2(X).start();
    }
}
```

FIG. 4.15 –

```
    public Barrier(int nThreads) {
        threads2Wait4 = nThreads;
    }

    public synchronized int waitForRest()
        throws InterruptedException {
        int threadNum = --threads2Wait4;
        if (iex != null) throw iex;
        if (threads2Wait4 <= 0) {
            notifyAll();
            return threadNum;
        }
        while (threads2Wait4 > 0) {
            if (iex != null) throw iex;
            try {
                wait();
            } catch(InterruptedException ex) {
                iex = ex;
                notifyAll();
            }
        }
        return threadNum;
    }

    public synchronized void freeAll() {
        iex = new InterruptedException("Barrier Released
                                        by freeAll");
        notifyAll();
    }
}
```

Fig. 4.16 –

```
public class ProcessIt implements Runnable {
    String[] is;
    Barrier bpStart, bp1, bp2, bpEnd;
    public ProcessIt(String[] sources) {
        is = sources;
        bpStart = new Barrier(sources.length);
        bp1 = new Barrier(sources.length);
        bp2 = new Barrier(sources.length);
        bpEnd = new Barrier(sources.length);
        for (int i=0;i<is.length; i++) {
            (new Thread(this)).start(); } }

    public void run() {
        try {
            int i = bpStart.waitForRest();
            doPhaseOne(is[i]);
            bp1.waitForRest();
            doPhaseTwo(is[i]);
            bp2.waitForRest();
            doPhaseThree(is[i]);
            bpEnd.waitForRest();
        } catch(InterruptedException ex) {};
    }

    public void doPhaseOne(String ps) {
    }

    public void doPhaseTwo(String ps) {
    }

    public void doPhaseThree(String ps) {
    }

    static public void main(String[] args) {
        ProcessIt pi = new ProcessIt(args);
    }
}
```

FIG. 4.17 –

encore que ces deux clients veulent en fait exactement les mêmes vols (ce qui est, pour des motifs temporels assez hasardeux...) et qu'il ne reste plus qu'une place sur chacun de ces vols. Comment faire en sorte de servir ces deux clients de façon cohérente?

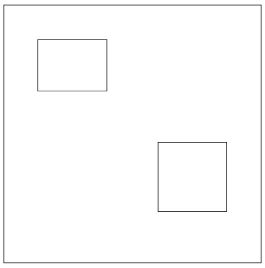
Regardons le programme implémentant les requêtes de Bill et Tom de la figure 4.19.

Le résultat est:

```
% java Serial
Bill: P(a)
Tom: P(b)
Bill: V(a)
Tom: V(b)
Bill: P(b)
Tom: P(a)
Bill: V(b)
Tom: V(a)
Le billet Paris->New York est au nom de: Bill
Le billet New York->Paris est au nom de: Tom
```

Ce qui n'est pas très heureux: en fait aucun des deux clients ne sera content car chacun pourra partir, mais pas revenir!

Il est facile de voir que le chemin d'exécution qui vient d'être pris correspond à un chemin sur la diagonale du graphe d'avancement:



qui "mélange" les transactions de nos deux utilisateurs.

Il existe plusieurs protocoles permettant d'atteindre un état plus cohérent de la base de données. Par exemple le protocole "2-phase locking" (voir [Abi00]) où chacun des deux clients essaie d'abord de verrouiller tous les billets voulus, avant d'y mettre son nom et de payer, puis de tout déverrouiller, permet cela. On dit que ce protocole est "séquentialisable", c'est à dire que tous les accès à la base de données distribuées se font comme si les deux clients étaient dans une même agence, et surtout dans une file ordonnée...

Quand on change dans `Requete`, l'ordre des verrous, dans la méthode `run()`, on obtient le programme de la figure 4.20 et l'exécution:

```
% java Serial2
Bill: P(a)
Bill: P(b)
Bill: V(a)
Bill: V(b)
Tom: P(b)
Tom: P(a)
Tom: V(b)
Tom: V(a)
Le billet Paris->New York est au nom de: Bill
Le billet New York->Paris est au nom de: Bill
```

```
public class Ticket {
    String nom;}

public class Requete extends Thread {
    Ticket Aller;
    Ticket Retour;
    Semaphore u;
    Semaphore v;
    public Requete(Ticket t, Ticket s,
                   Semaphore x, Semaphore y) {
        Aller = t;
        Retour = s;
        u = x;
        v = y; }

    public void run() {
        try {
            u.P();
            Thread.currentThread().sleep(100);
            if (Aller.nom == null)
                 Aller.nom = Thread.currentThread().
                                   getName();
            Thread.currentThread().sleep(100);
            u.V();
        } catch (InterruptedException e) {};
        try {
            v.P();
            Thread.currentThread().sleep(100);
            if (Retour.nom == null)
                Retour.nom = Thread.currentThread().
                                   getName();
            Thread.currentThread().sleep(100);
            v.V();
        } catch (InterruptedException e) {};
    }
}

public class Serial {
    public static void main(String[] args) {
        Semaphore a = new Semaphore(1,"a");
        Semaphore b = new Semaphore(1,"b");
        Ticket s = new Ticket();
        Ticket t = new Ticket();
        Requete Bill = new Requete(s,t,a,b);
        Requete Tom = new Requete(t,s,b,a);
        Bill.setName("Bill");
        Tom.setName("Tom");
        Bill.start();
        Tom.start();
        try {
            Tom.join();
            Bill.join();
        } catch(InterruptedException e) {};
        System.out.println("Le billet Paris->New York
                            est au nom de: "+s.nom);
        System.out.println("Le billet New York->Paris
                            est au nom de: "+t.nom); } }
```

FIG. 4.18 –

```
public void run() {
    try {
        Thread.currentThread().sleep(100);
        u.P();
        if (Aller.nom == null)
            Aller.nom = Thread.currentThread().
                                   getName();
        u.V();
        Thread.currentThread().sleep(100);
        Thread.currentThread().sleep(100);
        v.P();
        if (Retour.nom == null)
            Retour.nom = Thread.currentThread().
                                    getName();
        Thread.currentThread().sleep(100);
        v.V();
    } catch (InterruptedException e) {};
}
```

FIG. 4.19 –

```
public void run() {
    try {
        u.P();
        v.P();
        Thread.currentThread().sleep(100);
        if (Aller.nom == null)
            Aller.nom = Thread.currentThread().
                                   getName();
        Thread.currentThread().sleep(100);
        Thread.currentThread().sleep(100);
        if (Retour.nom == null)
            Retour.nom = Thread.currentThread().
                                    getName();
        Thread.currentThread().sleep(100);
        u.V();
        v.V();
    } catch (InterruptedException e) {};
}
```

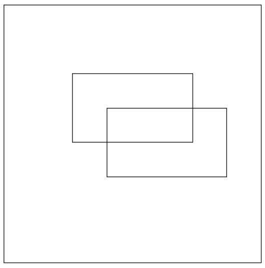
FIG. 4.20 –

qui ne satisfait vraiment que Bill mais qui au moins est cohérent!

Evidemment, il est aussi possible dans ce cas qu'une exécution donne également un interblocage:

```
% java Serial2
Bill: P(a)
Tom: P(b)
^C
```

On s'aperçoit que dans le cas de la figure 4.20, le graphe d'avancement interdit des chemins non séquentialisables, mais effectivement pas les points morts:



Les seules obstructions sont "au centre": on peut déformer tout chemin sur l'un des bords.

L'interface `Serializable` de JAVA, assure en fait cette propriété de séquentialisation:

– elle permet d'assurer la persistence de certains objets,
– elle est utilisée en particulier pour les entrées sorties

# Chapitre 5

# Algorithmes d'exclusion mutuelle (mémoire partagée)

## 5.1 Peut-on se passer de `synchronized`?

On cherche à programmer un mécanisme d'"exclusion mutuelle", en mémoire partagée, à partir des primitives que l'on a vu, sauf les primitives de synchronisation **`synchronized`** ainsi que **`wait`** et **`notify`**. On doit évidemment avoir une hypothèse de base sur la sémantique du système sous-jacent. On ne va imposer comme condition que le fait que la lecture et écriture en mémoire est **`atomique`** c'est à dire que plusieurs lectures et écritures en parallèle sont en fait exécutées séquentiellement dans un ordre à priori quelconque. Remarquer que c'est le cas en JAVA (on va le supposer en tout cas), au moins pour des types scalaires comme `int` mais pas `long`!

On veut donc pouvoir faire en sorte qu'un thread impose à tous les autres d'être l'unique à s'exécuter (pendant que les autres attendent leur tour). En fait, on pourra se déclarer content d'un algorithme d'exclusion mutuelle seulement s'il satisfait les conditions suivantes:

- (A) si un thread se trouve en section critique, tout autre thread doit être en train d'exécuter une section non-critique.
- (B) supposons qu'un groupe de thread $I$ exécute du code dans une section non-critique, et ne requièrt pas encore d'accéder à une section critique. Supposons maintenant que tous les autres processus (groupe $J$) requièrent l'accès à une section critique, mais qui n'en n'ont pas encore obtenu la permission. Alors le processus qui doit rentrer en section critique doit appartenir à $J$, et cette permission ne doit pas pouvoir être reportée indéfiniment.
- (C) un thread qui demande à entrer en section critique obtiendra toujours d'y rentrer au bout d'un temps fini (c'est une propriété d'*équité* qui permet d'éviter la *famine* d'un ou plusieurs processus).

La première idée qui vienne à l'esprit est de rajouter aux threads considérés un champ:

```
public class CS1 extends Thread {
    Thread tour = null;
    ...
}
```

qui précise quel thread doit s'exécuter. On veut écrire une méthode:

```
public void AttendtonTour()
```

qui attend l'autorisation de s'exécuter, et

```
public void RendlaMain()
```

qui autorise les autres threads présents à essayer de s'exécuter.

Pour simuler l'algorithme, on fait en sorte que tous les threads instance de `CS1` aient la même exécution:

```
public void run() {
    while(true) {
        AttendtonTour();
        System.out.println("C'est le tour de "+
                    Thread.currentThread().getName());
        RendlaMain();
    }
}
```

On en arrive au code naturel de la figure 5.1.

Mais cela contredit (A) par exemple. Il suffit de considérer une exécution synchrone des threads et on s'aperçoit que les deux processus peuvent rentrer dans leur section critique en même temps!

## 5.2 Premiers algorithmes?

Continuons sur l'idée de la dernière section, et essayons de programmer une première approximation de code permettant d'assurer l'exclusion mutuelle. Intéressons-nous d'abord à un premier cas dans lequel on a seulement deux processus $P_0$ et $P_1$. L'idée est d'assurer que l'on ne puisse jamais être en même temps dans certaine partie du code (appelée section critique) pour $P_0$ et $P_1$.

Le premier code auquel on puisse penser pour $P_i$ est ($i = 0, 1$, $j = (i + 1) \bmod 2$) est montré à la figure 5.2.

Remarquer que l'entier `turn` est déclaré en `volatile` c'est à dire qu'il est susceptible d'être modifié à tout moment par l'extérieur (un autre thread par exemple). Cela empêche `javac` de le placer dans un registre en cours d'exécution, pour optimiser le code (ce serait en l'occurrence une optimisation fausse dans notre cas!).

Ce code est une instance de la classe abstraite `ExclusionMutuelle` de la figure 5.3.

Pour tester cet algorithme, on utilise des taches très simples, simulant l'entrée et la sortie d'une section critique, voir figure 5.4.

Et enfin on utilise un `main` qui crée et lance deux tâches, voir figure 5.5.

Mais avec `Algorithme1`, les processus ne peuvent entrer en section critique qu'à tour de rôle, ce qui est contredit (B) (mais il est vrai que cet algorithme satisfait à (A) et (C), si on suppose que chaque section critique se déroule en temps fini).

En fait, on peut prouver (voir [Lyn96]) que pour résoudre le problème de l'exclusion mutuelle entre $n$ processus dans notre cadre, il faut au moins $n$ variables partagées distinctes, donc il est clair qu'il faut un peu compliquer le code précédent ($n = 2$, mais seulement une variable partagée, `turn`!).

Pour résoudre la difficulté précédente, il faut s'arranger pour connaître l'état de chaque processus afin de prendre une décision. Un essai de solution serait le code de la figure 5.6.

Si on remplace `INIT` par `false` on s'aperçoit que la condition (A) d'exclusion mutuelle n'est pas satisfaite!

```
public class CS1 extends Thread {
    Thread tour = null;

    public void AttendtonTour() {
        while (tour != Thread.currentThread()) {
            if (tour == null)
                tour = Thread.currentThread();
            try {
                Thread.sleep(100);
            } catch (Exception e) {}
        }
    }

    public void RendlaMain() {
        if (tour == Thread.currentThread())
            tour = null;
    }

    public void run() {
        while(true) {
            AttendtonTour();
            System.out.println("C'est le tour de "+
                   Thread.currentThread().getName());
            RendlaMain();
        } }

    public static void main(String[] args) {
        Thread Un = new CS1();
        Thread Deux = new CS1();
        Un.setName("UN");
        Deux.setName("DEUX");
        Un.start();
        Deux.start();
    } }
```

Fig. 5.1 –

```
public class Algorithme1 extends MutualExclusion
{
    public Algorithme1() {
        turn = 0;
    }

    public void Pmutex(int t) {
        while (turn != t)
            Thread.yield();
    }

    public void Vmutex(int t) {
        turn = 1-t;
    }

    private volatile int turn;
}
```

FIG. 5.2 –

```
public abstract class ExclusionMutuelle
{
    public static void Critique() {
        try {
            Thread.sleep((int) (Math.random()*3000));
        }
        catch (InterruptedException e) { }
    }

    public static void nonCritique() {
        try {
            Thread.sleep((int) (Math.random()*3000));
        }
        catch (InterruptedException e) { }
    }

    public abstract void Pmutex(int t);
    public abstract void Vmutex(int t);
}
```

FIG. 5.3 –

```
public class Tache extends Thread
{
    public Tache(String n,int i,ExclusionMutuelle s) {
        name = n;
        id = i;
        section = s;
    }

    public void run() {
        while (true) {
            section.Pmutex(id);
            System.out.println(name+" est en section critique");
            ExclusionMutuelle.Critique();
            section.Vmutex(id);

            System.out.println(name+" est sorti de la section critique");
            ExclusionMutuelle.nonCritique();
        }
    }

    private String name;
    private int id;
    private ExclusionMutuelle section;
}
```

Fig. 5.4 –

```
public class Test
{
    public static void main(String args[]) {
        ExclusionMutuelle alg = new Algorithme1();

        Tache zero = new Tache("Tache 0",0,alg);
        Tache un = new Tache("Tache 1",1,alg);

        zero.start();
        un.start();
    }
}
```

Fig. 5.5 –

```
public class Algorithme2 extends MutualExclusion
{
    public Algorithme2() {
        flag[0] = false;
        flag[1] = false;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = INIT;
        while (flag[other] == true)
            Thread.yield();
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile boolean[] flag = new boolean[2];
}
```

FIG. 5.6 –

Pour s'en convaincre il suffit de considérer l'exécution suivante:

| | | | |
|---|---|---|---|
| `zero` | `un` | flag[t] | flag[other] |
| `flag[1]=false`?: OUI | `flag[0]=false`?: OUI | false | false |
| `flag[0]:= true` | `flag[1]:=true` | true | true |
| `Section Critique` | `Section Critique` | - | - |

Si d'autre part, on fait `INIT=true`, la même exécution (synchrone) fait que les deux processus bouclent indéfiniment, contredisant (B) et (C)!

On propose alors le code de la figure 5.7, toujours plus compliqué...

Il réalise bien l'exclusion mutuelle mais peut causer un interblocage si les deux processus s'exécutent de façon purement synchrone. A ce moment là, après la première affectation, les deux tests de la boucle `while` sont vrais, d'où les deux `flag` sont mis à `false` *simultanément* puis à `true` et ainsi de suite. Les programmes de `zero` et de `un` bouclent avant la portion `Section Critique`.

Tout cela paraît sans issue. En fait, une solution est d'instaurer une règle de *politesse* entre les processeurs. C'est l'algorithme de Dekker (1965), à la figure 5.8.

## 5.3 Algorithme de Dekker

Il réalise l'exclusion mutuelle sans blocage mutuel. L'équité (la condition (C)) est vérifiée si l'ordonnanceur de tâche est lui-même équitable, ce que l'on supposera.

Considérons le code plus simple de la figure 5.9, proposé par Hyman en 1966.

Il est malheureusement faux car, si `turn` est à 0 et `zero` positionne `flag[1]` à `true` puis trouve `flag[0]` à `false`, alors *zero* met `flag[0]` à `true`, trouve `turn` égal à 0 et rentre en section critique. `un` affecte alors 1 à `turn` et rentre également en section critique!

Une vraie amélioration de l'algorithme de Dekker est l'algorithme de Peterson proposé en 1981.

```
public class Algorithme4 extends MutualExclusion
{
    public Algorithme4() {
        flag[0] = false;
        flag[1] = false;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;
        while (flag[other] == true) {
            flag[t] = false;
            while (flag[other])
                Thread.yield();
            flag[t] = true;
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile boolean[] flag = new boolean[2];
}
```

FIG. 5.7 –

## 5.4 Algorithme de Peterson

Contentons nous d'abord du cas simple où l'on dispose uniquement de deux tâches `zero` et `un`. Le code des deux tâches est donné à la figure 5.10.

On a bien alors exclusion mutuelle sans interblocage et avec équité (pas de famine).

Donnons ici quelques idées de preuve de cet algorithme.

On va décorer le code par des "assertions" écrites avec des prédicats logiques du premier ordre, portant sur les valeurs des variables du programme (et donc également de celles que l'on vient d'introduire), avec comme prédicats l'égalité avec des constantes (cela nous suffira ici). Ces assertions seront introduites entre deux instructions consécutives, c'est à dire qu'elles seront associées à chaque point de contrôle de chaque processus.

On va introduire, de façon intuitive et pas trop formelle, une méthode de raisonnement et de preuve sur des processus parallèles (disons, en nombre $n$) en mémoire partagée due à l'origine à S. Owicki et D. Gries [OG75].

Les assertions que l'on rajoute doivent être des "invariants" décrivant des relations logiques entre les valeurs des variables, vraies pour toutes les exécutions possibles (en particulier tous les ordonnancements des actions atomiques). Cela veut dire la chose suivante. Soit $T$ le système de transition donnant la sémantique du programme considéré. On sait qu'un état de $T$ est un (n+1)-uplet $(c_1, \ldots, c_n, E)$ où $c_i$ $(1 \leq i \leq n)$ est un point de contrôle du $i$ème processus et $E$ est un environnement, c'est à dire une fonction qui à chaque variable du programme associe une valeur. Alors le système d'assertion $A$ qui à chaque point de contrôle $c$ (de n'importe quel processus $i$) associe un prédicat $P$ est admissible si quelque soit l'état $(c_1, \ldots, c_n, E)$ du système de transition tel que $c_i = c$ alors $P$ est vrai pour l'environnement $E$.

```
public class Dekker extends ExclusionMutuelle
{
    public Dekker() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;

        while (flag[other] == true) {
            if (turn == other) {
                flag[t] = false;
                while (turn == other)
                    Thread.yield();
                flag[t] = true;
            }
}
    }

    public void Vmutex(int t) {
        turn = 1-t;
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

FIG. 5.8 –

```
public class Hyman extends ExclusionMutuelle
{
    public Hyman() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }

    public void Pmutex(int t) {
        int other = 1-t;
        flag[t]= true;

        while (turn==other) {
            while (flag[other])
                Thread.yield();
            turn = t;
        }
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

FIG. 5.9 –

```
public class Peterson extends MutualExclusion
{
    public Peterson() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;
        turn = t;
        while (flag[other] && (turn == t))
            Thread.yield();
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

FIG. 5.10 –

L'idée est que l'on veut pouvoir inférer quand c'est possible, ou tout du moins vérifier que les assertions que l'on rajoute dans le code de chacun des processus dit bien des choses vraies sur toutes les exécutions possibles du code.

Soit $I$ une instruction ou un block d'instructions JAVA. On rajoute des assertions décrivant ce qui se passe avant et après le code $I$ en écrivant $\{p\}I\{q\}$. On a les règles suivantes, pour un fragment de JAVA, qui va nous suffire pour le moment:

–
$$\{p[x \leftarrow u]\}x = u\{p\}$$

–
$$\{p_1\}\left(if\ B\ then\ \{p_2\}I_1\{q_2\}\ else\{p_3\}I_2\{q_3\}\right)\{q_1\}$$

si on a,

$$\begin{array}{l} p_1 \wedge B \Rightarrow p_2 \\ p_1 \wedge \neg B \Rightarrow p_3 \\ \{p_2\}I_1\{q_2\} \\ \{p_3\}I_2\{q_3\} \\ q_2 \Rightarrow q_1 \\ q_3 \Rightarrow q_1 \end{array}$$

–
$$\{p_1\}whileB\left\{\{p_2\}C\{q_2\}\right\}\{q_1\}$$

si on a,

$$\begin{array}{l} p_1 \wedge B \Rightarrow p_2 \\ p_1 \wedge \neg B \Rightarrow q_1 \\ \{p_2\}C\{q_2\} \\ q_2 \wedge B \Rightarrow p_2 \\ q_2 \wedge \neg B \Rightarrow q_1 \end{array}$$

Par exemple (voir [Win93]), le système suivant est un système d'assertions correctes:

```
{ x=x0 et y=y0 }
q = 0;
while (x >= y) {
  {x=x0-qy0 et y=y0}
  q = q+1;
  {x=x0-(q-1)y0 et y=y0}
  x = x-y;
  {x=x0-qy0 et y=y0}
}
{x=x0-qy0 et x<y et y=y0}
```

Une fois la preuve faite séquentiellement, il faut prouver que les invariants trouvés pour chaque thread pris séparément, est encore valide si tout ou partie d'autres threads s'exécutent entre deux instructions, modifiant ainsi l'état local du thread! Cela s'appelle la preuve de non-interférence. Attention car on suppose ici que les affectations sont atomiques (ce sera vrai en JAVA dans la suite pour les affectations de constantes, ce qui sera le seul cas intéressant pour nous; sinon il faudrait décomposer les affectation en *load*, *store* etc. comme expliqué précédemment).

Pour que cela puisse se faire de façon correcte, il faut introduire des variables nouvelles dans le code séquentiel, qui déterminent le point de contrôle auquel le processus est arrivé. Sans cela, la méthode est fausse (en fait, l'article d'origine de S. Owicki et de D. Gries comportait cet oubli). Par exemple, on devrait transformer la preuve précédente en:

```
{ x=x0 et y=y0 et c=0}
q = 0;
{ x=x0 et y=y0 et c=1}
while (x >= y) {
  {x=x0-qy0 et y=y0 et c=2}
  q = q+1;
  {x=x0-(q-1)y0 et y=y0 et c=3}
  x = x-y;
  {x=x0-qy0 et y=y0 et c=4}
}
{x=x0-qy0 et x<y et y=y0 et c=5}
```

`c` représente le point de contrôle courant, numéroté de façon fort logique de 1 à 5 ici.

On va donc supposer que chaque thread $P_i$ parmi $P_1, \ldots, P_k$ a un compteur `ci` et des assertions $A_{i,k}$ avant les instructions $I_{i,k}$ de $P_i$ (pour $k$ variant sur l'ensemble des points de contrôle de $P_i$) utilisant éventuellement des prédicats aussi sur les autres compteurs ordinaux `c1`,. . .,`ck`. Alors il faut vérifier les conditions dites de non-interférences suivantes, quand $I_{j,l}$ est une affectation `x=u`:

$$\{(A_{i,k} \wedge A_{j,l})[x \leftarrow u]\} I_{j,l} \{A_{i,k}\}$$

On peut simplifier un peu cela dans le cas qui nous préoccupe, car on n'a qu'une affectation sur un objet partagé, donc on n'a pas besoin de numéroter toutes les instructions. Introduisons deux variables auxiliaires `after[0]` et `after[1]` (qui servent à indiquer si le contrôle est après `turn=t` dans le code). Rajoutons également les instructions `after[0]=true`, `after[1]=true` après respectivement `turn=0` et `turn=1`.

```
{ non flag[0] }
flag[0]=true; after[0]=false;
{ flag[0] et non after[0] }
turn=0; after[0]=true;
{ inv: flag[0] et after[0] et I }
while (flag[1] && (turn==0))
   { flag[0] et after[0] et I }
   Thread.yield();
{ flag[0] et after[0] et (flag[1] et (non(after[1]) ou turn=1)) }
[ Section Critique CS1]
{ flag[0] }
flag[0]=false;
```

FIG. 5.11 –

```
{ non flag[1] }
flag[1]=true; after[1]=false;
{ flag[1] et non after[1] }
turn=1; after[1]=true;
{ inv: flag[1] et after[1] et I }
while (flag[0] && (turn==0)) do
   { flag[1] et after[1] et I }
   Thread.yield();
{ flag[1] et after[1] et (flag[0] et (non(after[0]) ou turn=0)) }
[ Section Critique CS2]
{ flag[1] }
flag[1]=false;
```

FIG. 5.12 –

En utilisant l'abbréviation `I= [turn=1 ou turn=2]` on commente le code avec les assertions comme à la figure 5.11.

De même pour le deuxième processus, à la figure 5.12.

Il est relativement aisé de se convaincre ces assertions forment bien un shéma de preuve correct de chacun des processus pris séquentiellement. Seule subsiste la preuve de non-interférence. En fait, seul,

```
pre(CS1) = flag[0] et after[0] et (flag[1] et (non(after[1])
                                                ou (turn=1)))
```

contient des références à des objets manipulés par l'autre processus, et ce, seulement en les lignes

```
flag[1]=true; after[1]=false;
```

et

```
turn=1; after[1]=true;
```

Or,

```
{ pre(CS1) } flag[1]=true; after[1]=false; { pre(CS1) }
```

et

```
{ pre(CS1) } turn=1; after[1]=true; { pre(CS1) }
```

```
public class PetersonN extends ExclusionMutuelle
{
    public PetersonN(int nb) {
        int i;
        nproc = nb;
        turn = new int[nproc-1];
        flag = new int[nproc];
        for (i=0; i < nproc; i++) {
            flag[i] = -1; }
    }

    public void Pmutex(int t) {
        int j, k;
        boolean cond;
        for (j=0; j < nproc-1; j++) {
            flag[t] = j;
            turn[j] = t;
            cond = true;
            for (k=0; (k < nproc) && (k != t); k++)
                cond = cond || (flag[k] >= j);
            while (cond && (turn[j] == t))
                Thread.yield();
        }
    }

    public void Vmutex(int t) {
        flag[t] = -1;
    }

    private volatile int[] turn;
    private int nproc;
    private volatile int[] flag;
}
```

FIG. 5.13 –

et de même par symétrie. De plus, on peut voir que

```
pre(CS1) et pre(CS2) implique (turn=0 et turn=1)
```

Ce qui implique que l'on a toujours

```
non(pre(CS1) et pre(CS2))
```

prouvant ainsi l'exclusion mutuelle.

Remarquez qu'il existe une généralisation de l'algorithme de Peterson à $n$ processeurs. C'est le programme de la figure 5.13, codé sur chacun des $n$ processeurs ($liste[0]$ à $liste[nproc-1]$).

Il faut évidemment légèrement changer le programme de test (ici pour 5 tâches), voir figure 5.14.

En fait c'est la solution pour 2 processeurs itérée $n-1$ fois. `flag[i]`$=-1$ signifie que $liste[i]$ ne s'engage pas en section critique. `turn` permet de gèrer les conflits dans un couple de processus. Cet algorithme réalise bien l'exclusion mutuelle sans interblocage et avec équité.

```
public class TestN
{
    public final static int nb = 5;

    public static void main(String args[]) {
        int i;
        Tache[] liste;

        liste = new Tache[nb];
        ExclusionMutuelle alg = new PetersonN(nb);

        for (i=0; i < nb; i++) {
            liste[i] = new Tache("Tache "+i,i,alg);
            liste[i].start();
        }
    }
}
```

Fig. 5.14 –

Dans les améliorations possibles, il y a en particulier l'algorithme de Burns (1981). C'est une solution entièment symétrique qui minimise le nombre de variables utilisées ainsi que les valeurs qu'elles peuvent prendre en cours de programme.

# Bibliographie

[Abi00] S. Abiteboul. Bases de données, 2000.

[Bou00] F. Bourdoncle. Systèmes et réseaux, 2000.

[CS98] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan-Kaufmann, 1998.

[Dij68] E.W. Dijkstra. *Cooperating Sequential Processes*. Academic Press, 1968.

[GNS00] E. Goubault, F. Nataf, and M. Schoenauer. Calcul parallèle, 2000.

[Hoa74] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communication of the ACM*, 17, 1974.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[OG75] Susan S. Owicki and David Gries. Proving properties of parallel programs: An axiomatic approach. Technical Report TR75-243, Cornell University, Computer Science Department, May 1975.

[OW00] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 2000.

[Ray97] M. Raynal. *Algorithmique et Parallélisme, le problème de l'exclusion mutuelle*. Dunod, 1997.

[Rem00] D. Remy. Cours de compilation, 2000.

[Rob00] Y. Robert. Algorithmique parallèle, 2000.

[Win93] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.