9

Les processus

Ainsi que nous avons pu le mentionner, un système Squeak comprend une machine virtuelle, qui met en œuvre un processeur virtuel ; ce processus exécute les instructions (exprimées en *bytecode*) qui correspondent au code des méthodes Squeak.

Dans les mécanismes dont nous avons traités jusqu'ici, il n'existait qu'un seul flux de programme, s'exécutant en mémoire. Une nouvelle notion va permettre d'introduire un contrôle spécifique sur le flux des exécutions au sein du système : une portion de code peut être interrompue (par le message `suspend`) et reprise plus tard (`resume`). Si plusieurs portions de code sont en attente en mémoire à un instant donné, le choix quant à celle qu'il faut reprendre est effectué par un objet gestionnaire, principalement sur la base de priorités.

Ces portions de code s'appellent des processus (instances de la classe `Process`, appartenant à la catégorie `Kernel-Processes`) et permettent de simuler le fonctionnement simultané de plusieurs programmes en mémoire.

Qu'est ce qu'un processus ?

Les processus sont utilisés en particulier dans les applications multi-utilisateurs (serveurs de bases de données, serveurs Web…) pour traiter « simultanément » plusieurs demandes ; dans le domaine de la simulation, ce sera pour étudier le fonctionnement d'un système concurrent (chaîne de production, systèmes multi-agents…), tandis que dans tout système d'exploitation ils seront mis à profit pour permettre l'utilisation simultanée de plusieurs applications par l'utilisateur, ainsi que la gestion des différents périphériques.

Dans la mesure où il n'existe qu'un seul processeur matériel (du moins sur les machines mono-processeur), la simultanéité ne sera pas réelle et il s'agira nécessairement de temps partagé : chaque processus dispose du processeur pendant un certaine période, déterminée par l'attribution d'un quantum de temps prédéfini pour chaque processus, ou par l'exécution d'une instruction redonnant explicitement le contrôle au gestionnaire de processus ; ce dernier, implémenté par la classe `ProcessorScheduler` et son unique instance `Processor` peut alors donner la main à

un autre processus(voir figure 9-1). Cette seconde solution est utilisée en standard dans Squeak, bien que la première puisse être mise en œuvre sans trop de difficultés.

Remarque

Il suffit de créer un processus de très haut niveau, dont le fonctionnement est contrôlé par l'horloge avec un délai d'attente fixé (*time slice,* qui donne son nom à ce mécanisme de *time slicing*). Puisque ce processus est de très haute priorité, il aura nécessairement la main lorsque son délai d'attente viendra à expiration. Il lui suffit alors d'envoyer un message qui contraigne le processus « courant » (celui qui était actif avant lui) à rendre la main (message `yield`) pour permettre aux autres processus de s'exécuter.

Techniquement parlant, les processus Squeak sont des processus légers (threads), préemptifs d'un niveau de priorité à l'autre (un processus de niveau supérieur peut prendre la main « de force » sur un processus de niveau inférieur), et coopératifs entre processus de niveau de priorité identiques (un processus doit rendre la main volontairement pour que les autres aient une chance de s'exécuter).

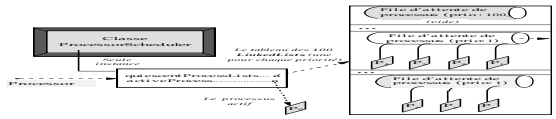

Figure 9-1. Le gestionnaire de processus (seule instance de ProcessorScheduler) et ses listes de processus en attente (P0 est de priorité i, i étant supposée être ici la priorité la plus élevée correspondant à une liste non vide)

Un processus est une instance de `Process`, sous-classe de `Link` (abordée au chapitre 6 traitant des collections). Nous verrons en effet que la gestion des processus implique qu'ils puissent être stockés (alternativement) dans différents types de listes chaînées, et que ces listes ont pour éléments des `Link`.

Un processus est pour l'essentiel caractérisé par ses variables d'instance :

- `priority` : une valeur entière utilisée pour ordonner les processus par niveaux de priorité ;
- `suspendedContext` : le contexte actif au moment de la suspension, c'est-à-dire un ensemble d'informations pertinentes au moment de l'exécution du processus, relatives au contexte qui est à l'origine du contexte courant, du receveur de ce contexte, des arguments et informations temporaires sur la pile de travail… ;
- `myList` : la liste de processus où se trouve le processus suspendu (il peut y avoir plusieurs listes de ce genre, et ce en liaison également avec la notion de sémaphore…).

Remarque terminologique

Le terme de *processus* est employé en Squeak en raison du nom de la classe `Process`.

En toute rigueur, il ne s'agit pas de processus, mais de threads, ou processus légers. Un processus, au sens par exemple des processus Unix (créés par `fork()`), s'exécute en effet dans un segment de données totalement autonome. Il ne peut donc communiquer avec les autres processus que par des « tubes » (pipes), ou par des segments de mémoire partagés qui doivent être explicitement définis.

Au contraire, les threads créés à partir d'un même processus partagent l'espace de données dont ils sont issus, et peuvent donc communiquer par son intermédiaire. C'est le cas en Squeak, où les processus, qui sont donc

des threads, peuvent accéder aux objets de l'espace de nommage à partir duquel ils ont été créés (variables globales, classes…).

Création de processus

On crée un processus en envoyant un message à un bloc (une instance de la classe `BlockContext`, par exemple `[2+4] fork`). Plusieurs messages de création sont définis :

- `newProcess` crée un processus dont le code est celui du bloc receveur, mais qu'il n'insère dans aucune file des processus en attente. La priorité qui est affectée au nouveau processus est celle du processus actif ;
- `newProcessWith:` permet de créer un nouveau processus en transmettant des arguments au bloc receveur ; cela permet de définir le processus de façon générale, sans référence aux arguments qui sont définis à l'extérieur de ce bloc (plus précisément, il est ainsi possible de *ne pas* considérer sous forme d'un objet le contexte extérieur au bloc, ce qui améliore les performances) ;
- `fork` permet de créer un processus (avec `newProcess`) et de le « planifier » (*schedule*). Un message `resume` lui est envoyé,ce qui le positionne dans la file d'attente correspondant à sa priorité et le rend susceptible d'être activé ;
- `forkAt:` effectue la même opération que `fork`, mais permet de préciser le niveau de priorité à affecter au processus créé.

Priorités des processus

À un moment donné, plusieurs processus peuvent donc se trouver en mémoire, l'un d'eux seulement étant actif. Le contrôle de l'exécution des processus est la tâche de l'unique instance de la classe `ProcessorScheduler`, stockée dans une variable globale du système, la variable `Processor`. Pour déterminer la séquence d'activation des processus, et en permettre une gestion fine, une priorité est affectée à chacun d'eux, qui peut varier de 1 à 10.

Certains de ces niveaux de priorité ont été dénommés et correspondent à des situations répertoriées (`BackgroundProcess`, `HighIOPriority`, `LowIOPriority`, `SystemBackgroundPriority`, `SystemRockBottomPriority`, `TimingPriority`, `UserBackgroundPriority`, `UserInterruptPriority`, `UserSchedulingPriority`).

Tableau 9-1. Rôles des différentes priorités des processus et méthodes d'accès

Valeur de la priorité	Méthode	But
8	`timingPriority`	Processus devant fonctionner en temps réel
7	`highIOPriority`	Entrées-sorties critiques (par exemple réseau)
6	`lowIOPriority`	Entrées-sorties normales (par exemple clavier)

5	`userInterruptPriority`	Interaction utilisateur de haut niveau de priorité
4	`userSchedulingPriority`	Interaction utilisateur normale
3	`userBackgroundPriority`	Processus utilisateurs en tâche de fond
2	`systemBackgroundPriority`	Processus système en tâche de fond
1	`systemRockBottomPriority`	Processus système de plus bas niveau de priorité

On obtient la valeur numérique de la priorité correspondant à un nom en envoyant à l'instance `Processor` le message indiqué dans la deuxième colonne du tableau 9-1. Par exemple, `Processor lowIOPriority` renvoie `6`.

Les différents processus qui doivent s'exécuter (par exemple, ceux qui ont reçu un message `resume`, mais qui n'ont pas une priorité suffisante pour être *le* processus actif) sont stockés dans des listes, correspondant à leur priorité, où ils attendent que le gestionnaire les active réellement. Ils sont alors qualifiés de processus activables.

Le gestionnaire (`Processor`) donne toujours le contrôle au processus en attente qui a le plus haut niveau de priorité. Lorsque plusieurs processus de même niveau de priorité sont en attente, pour que les autres puissent s'exécuter, le processus actif doit redonner la main en exécutant `Processor yield`, ce qui provoque le renvoi du processus actif à la fin de la file des processus en attente qui ont le même niveau de priorité.

Plusieurs méthodes d'instance sont définies sur les processus pour gérer les changements d'état et de priorité :

- `priority` et `priority:` permettent d'obtenir ou d'affecter la priorité du processus receveur ;
- `suspend` arrête temporairement le processus receveur, en préservant son contexte ;
- `resume` permet de rendre à nouveau `activable` un processus qui avait été suspendu ;
- `terminate` arrête définitivement le processus receveur.

D'autres états (notamment l'état d'attente) sont dus à l'utilisation d'un sémaphore, notion que nous allons approfondir à présent.

Les sémaphores

Les sémaphores sont des mécanismes de synchronisation entre processus qui permettent de protéger des données ou des ressources utilisées par plusieurs processus. Ils permettent, en particulier, de mettre en attente un processus qui demande une certaine ressource, tant que cette dernière n'est pas disponible, et ce, sans que le processus doive vérifier périodiquement la disponibilité de la ressource demandée.

Un sémaphore peut être créé en envoyant le message `new` à la classe `Semaphore` (d'autres méthodes de création sont possibles, par exemple `forMutualExclusion`, qui crée un processus avec un signal d'avance). Lorsqu'un processus souhaite utiliser la ressource protégée par le sémaphore, il lui envoie un message ; les méthodes de la ressource doivent donc être conçues de

telle façon que cela provoque l'envoi du message `wait` au sémaphore pertinent (un exemple de ce mécanisme est fourni avec les `SharedQueue`, abordées au chapitre 6 consacré aux collections). Si le nombre de processus connectés à ce moment au sémaphore est inférieur au nombre de processus qui sont autorisés à se connecter, ce message n'est pas bloquant et le processus peut continuer ses opérations en utilisant la ressource comme il le souhaitait. Dans le cas contraire, le processus est stocké en fin de la file d'attente du sémaphore, et ne sera relancé que lorsque les processus qui le précèdent auront libéré la place.

Un processus signifie à un sémaphore qu'il libère la ressource en lui envoyant le message `signal`. Le sémaphore envoie alors un message `resume` au premier processus en attente de sa file d'attente, ce qui rend ce dernier activable, c'est-à-dire qu'il est ajouté dans la file d'attente du `ProcessorScheduler` correspondant à sa priorité (voir figure 9-2 ci-après).

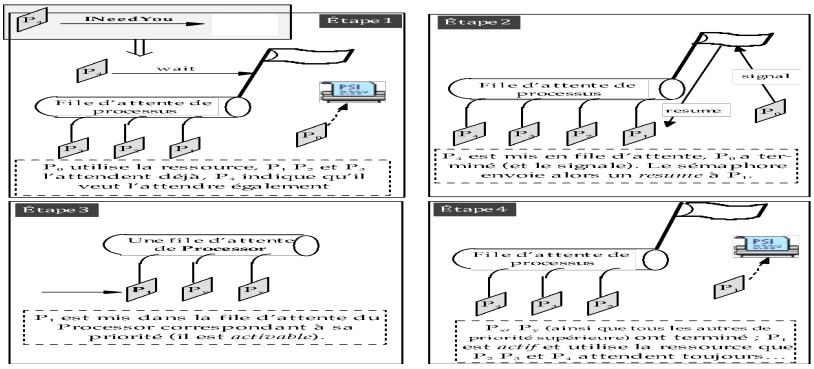

Figure 9-2. Fonctionnement d'un sémaphore gérant l'accès à une ressource

Différents états d'un processus

Par souci de synthèse, nous allons préciser les différents états dans lesquels un processus peut se trouver à un instant donné : suspendu, activable, actif ou en attente (voir figure 9-3 ci-après).

- *actif* : il est en cours d'exécution ;
- *activable* : il est dans une des files d'attente de `Processor` ;
- *en attente* : il est suspendu sur un sémaphore (il se trouve dans la file d'attente du sémaphore ; il n'est donc pas encore activable) ;
- *suspendu* : si c'est le processus actif, il est interrompu et pourra être réactivé par la suite ; sinon, il est ôté de la file d'attente des processus activables où il se trouve.

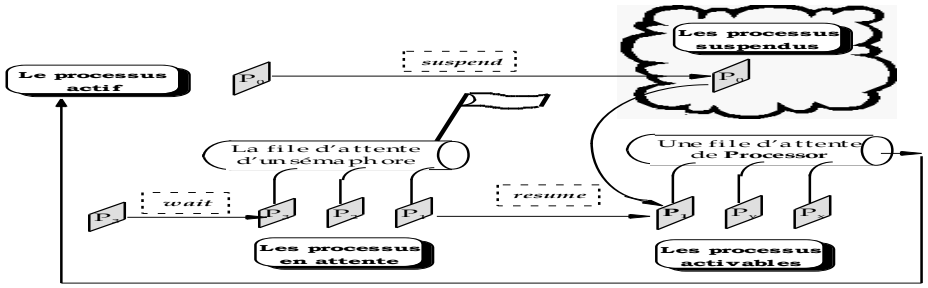

Figure 9-3. Différents états des processus et les messages qui font passer de l'un à l'autre

Les mécanismes de temporisation : la classe `Delay`

Les instances de la classe `Delay` sont utilisées pour mettre en attente un processus durant un certain temps.

Un `Delay` (sous-classe de `Object`) peut être créé en spécifiant une durée (`forMilliseconds:`, `forSeconds:`). Il est principalement constitué d'un sémaphore (stocké dans la variable d'instance `delaySemaphore`) sur lequel le processus actif est mis en attente lorsque l'instance de `Delay` ainsi créée reçoit le message `wait`. Ce message consiste pour l'essentiel en l'envoi de `wait` au sémaphore `delaySemaphore`, et en l'envoi du message `resume` au même sémaphore lorsque le délai spécifié est venu à expiration.

Dans l'exemple suivant, nous utilisons les processus et les délais pour afficher une horloge qui signale l'heure dans le `Transcript` toutes les trois secondes.

Nous créons pour cela une classe `Horloge`. Chaque horloge a un processus (`process`), qui a pour seule fonction de notifier un changement toutes les trois secondes, et une variable booléenne (`stop`) utilisée pour arrêter le processus.

```
Object subclass: #Horloge
      instanceVariableNames: ' process stop '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Exemples'
```

Nous définissons les trois méthodes d'instance suivantes :

```
start
      process resume.
```

```
stop
      stop := true.
      Transcript cr; show: 'l''horloge est arrêtée'
```

```
initialize
      process := [| delay |
            delay := Delay forSeconds: 3.
            [stop]
```

```
                whileFalse: [
                   delay wait.
                   Transcript cr; show: Time now printString]] newProcess.
      process priority: Processor timingPriority.
      stop := false
```

La boucle sans fin du processus de l'horloge (ligne 5 de la méthode `initialize`) est temporisée par l'envoi du message `wait` (ligne 6) selon le délai qui a été créé en ligne 3, avec une durée d'attente de trois secondes.

Quelques exemples d'utilisation des processus

Les quelques exemples suivants illustrent différents aspects des processus.

Le message `fork` envoyé aux blocs présentés ci-après crée deux processus P1 et P2 qui effectuent chacun dix affichages dans le `Transcript` (respectivement, les nombres de 1 à 10 pour le premier, de 11 à 20 pour le second). Ces deux processus sont créés avec la même priorité.

```
[1   to: 10 do: [:i| Transcript show:' '; show:  i printString. ]] fork. "(P1)"
[11 to: 20 do: [:i| Transcript show:' '; show:  i printString. ]] fork.  "(P2)"
```

L'affichage dans le `Transcript` (`1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20`) montre alors qu'aucun des deux ne redonne le contrôle au processeur durant sa boucle :le premier s'exécute jusqu'à sa fin, puis le second fait de même. Cela prouve, en outre, qu'aucun quantum de temps n'est attribué par défaut aux processus.

Redonner la main : le message `yield`

Pour que les exécutions des processus puissent alterner, nous devons introduire `Processor yield` dans chacune des boucles, ce qui entraîne l'interruption du processus actif et permet à d'autres processus de même priorité de s'exécuter (à noter l'utilisation de `newProcess` puis de `resume`, autre manière de lancer des processus) :

```
      p1 :=  [1   to: 10 do: [:i| Transcript show:' '; show:  i printString. Processor
yield]. ] newProcess.
      p2 :=  [11 to: 20 do: [:j| Transcript show:' '; show:  j printString.  Processor
yield]. ] newProcess .
      p2 resume. p1 resume
```

Nous obtenons alors l'affichage `11 1 12 2 13 3 14 4 15 5 16 6 17 7 18 8 19 9 20 10`, qui montre que les processus se sont bien déroulés de façon « quasi parallèle ».

La méthode `yield` est très élégamment implémentée de la façon suivante :

```
yield
  | semaphore |
  semaphore := Semaphore new.
  [semaphore signal] fork.
  semaphore wait
```

Dans le cas où seraient présents des processus activables de même priorité que le processus actif, un sémaphore nouveau est créé (ligne 3), ainsi qu'un nouveau processus (ligne 4 – nous l'appellerons $Px_{2,}$, voir les étapes de la figure 9-4) dont le rôle consiste à envoyer le message

`signal` de libération de la ressource au nouveau sémaphore (cela signifie que le processus est créé mais non pas qu'il envoie ce message : ligne 5, P1 est mis en attente et redeviendra activable quand Px_1 sera rendu actif, et donc quand tous les processus avant P x_1 auront été terminés ou (re)mis en attente). Le fonctionnement de l'ensemble est synthétisé dans les différentes étapes de la figure 9-4 ci-après :

Figure 9-4. Les diverses séquences produites par l'instruction `yield` lors d'un affichage de nombres par deux processus

Gestion des priorités

Remarque importante

Les processus qui ont une priorité supérieure à 5 ont priorité sur les interactions de l'utilisateur. Ces processus dominent donc les actions de ce dernier, ce qui peut rendre impossible l'utilisation des menus ou du clavier.

Il faut noter également que les processus de priorité supérieure ou égale à 7 prennent le pas sur les processus de gestion des fenêtres, et doivent donc être de durée limitée, sauf à geler l'interface le temps de leur exécution.

Dans l'exemple suivant, nous créons un processus de priorité basse (`3`), qui affiche les nombres de `1` à `50` dans le `Transcript`. Pendant le déroulement de ce processus, l'utilisateur exécute de temps en temps l'expression `Transcript show: 'coucou'`, qui est affichée immédiatement. Cela montre que l'utilisateur peut continuer à interagir avec le système pendant le déroulement du premier processus :

```
 [1 to: 50 do: [:i| Transcript show:' '; show:  i printString. ]] forkAt: (Processor
userBackgroundPriority).
```

Nous obtenons alors l'affichage suivant :

```
 1 2 3 4 5 6 7 8 9 10 11 12 13  coucou 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 coucou 32 33 34 35 36 37 38 39 40 41 42
43 44 45 coucou 46 47 48 49 50
```

Avertissement

Attention, des problèmes peuvent survenir lors de l'utilisation simultanée du `Transcript` par plusieurs processus qui effectuent des affichages.

En effet, les méthodes d'affichage ne sont pas protégées contre les accès concurrents, et peuvent être interrompues à des moments inopportuns. Il peut alors être nécessaire de définir un `Transcript` particulier (sous-classe de `Transcript`), protégé contre les accès concurrents.

Ce problème se pose d'ailleurs pour toute ressource partageable, et nécessite l'utilisation de sémaphore d'exclusion mutuelle et de sections critiques.

Ainsi que nous l'avons souligné plus haut, un processus activable d'un niveau de priorité donné est toujours activé avant un autre processus activable de niveau de priorité inférieur, comme le montre l'exemple suivant :

```
[10 timesRepeat: [Transcript show:' '; show: '3'. Processor yield]] forkAt: 2.
[10 timesRepeat: [ Transcript show:' '; show: '2'. Processor yield]] forkAt: 3.
[10 timesRepeat: [ Transcript show:' '; show: '1'. Processor yield]] forkAt: 4.
```

Le troisième processus, de priorité 4, est exécuté avant le deuxième (priorité 3), lui-même exécuté avant le premier (priorité 2), comme le montre l'affichage suivant :

```
1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
```

Dans bien des situations, l'usage de simples priorités ne suffit pas : les processus doivent être explicitement coordonnés. Dans l'exemple suivant, nous utilisons un sémaphore pour synchroniser l'exécution de deux processus : nos contraintes sont que `p1` doit être le premier à commencer son exécution, mais qu'il doit ensuite attendre que `p2` ait effectué une action donnée pour pouvoir poursuivre.

Il faut veiller à ce que la priorité de `p1` soit supérieure à celle de `p2` pour que le premier se mette en attente (`wait`) sur le sémaphore que le second débloquera au moment voulu (`signal`).

```
|s p1 p2|
s := Semaphore new.
p1 := [
     Transcript cr; show: 'P1 : je suis le premier'; cr.
     s wait.
     Transcript cr; show: 'P1 : je suis débloqué'; cr.] newProcess priority: 4.
p2 := [
     Transcript show: 'P2 : j''envoie signal au sémaphore S'; cr.
      s signal.
     Transcript  show: 'P2 : j''ai envoyé signal à S'.] newProcess priority: 3.
```

```
p1 resume. p2 resume.
```

Nous obtenons alors les affichages suivants, qui montrent que `p2` s'interrompt bien pour laisser `p1` s'exécuter : `p1`, de priorité supérieure à `p2`, a été activé dès que ce dernier a envoyé le message `signal` au sémaphore, `p2` recommençant son exécution dès que `p1` a fini la sienne.

```
P1 : je suis le premier
P2 : j'envoie signal au sémaphore S

P1 : je suis débloqué
P2 : j'ai envoyé signal à S
```

L'importance des priorités transparaît dans le code suivant (où les priorités ont été échangées), et qui ne fonctionne pas de la façon voulue, car `p2` conserve le contrôle jusqu'à la fin de son exécution, `p1` n'étant activé qu'à ce moment-là, comme le montrent les affichages.

```
|s p1 p2|
s := Semaphore new.
p1 :=
      [s wait.
     Transcript cr; show: 'P1 : je suis débloqué'; cr.] newProcess priority: 42.
p2 :=
     [Transcript show: 'P2 : j''envoie signal au sémaphore S'; cr.
      s signal.
     Transcript  show: 'P2 : j''ai envoyé signal au sémaphore S'.] newProcess priority:
43.
p1 resume.
p2 resume.
```

Voici l'affichage qui en résulte :

```
P2 : j'envoie signal au sémaphore S
P2 : j'ai envoyé signal au sémaphore S
```

Un grand classique de la synchronisation de processus : le dîner des philosophes

Le dîner des philosophes est un excellent modèle pour un grand nombre de problèmes réels.

Il s'agit d'un problème de synchronisation inter-processus, qui permet de mettre en évidence deux problèmes fondamentaux : la coordination des processus entre eux, et les problèmes d'inter-blocages (*dead-lock* en anglais), lorsque deux processus se bloquent mutuellement, chacun attendant que l'autre libère une ressource pour pouvoir redémarrer.

Présentation du problème

Le problème du dîner est le suivant : 5 philosophes sont réunis autour d'une table ronde pour dîner et se détendre tout en philosophant. Le repas est constitué de spaghettis. Pour qu'un philosophe puisse manger, il doit utiliser deux couverts (afin d'enrouler ses spaghettis selon les règles de bienséance en vigueur dans les cercles de philosophie). La difficulté est qu'il n'y a que cinq couverts sur la table (voir figure 9-5 ci-après) :

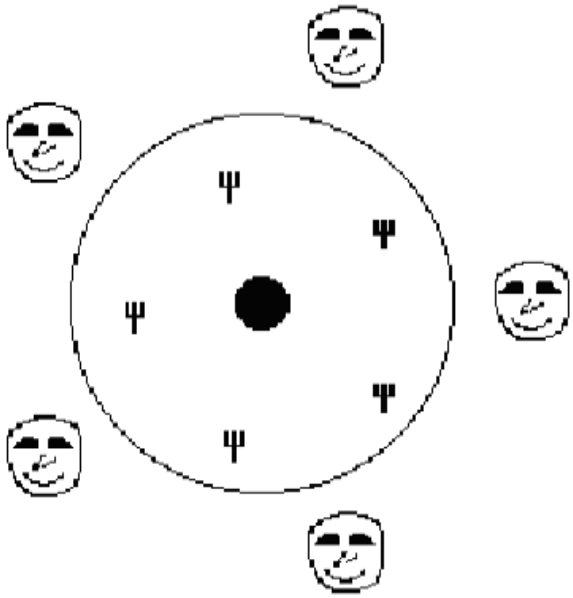

Figure 9-5. La situation à la table des philosophes

Bien entendu, dans un cercle de philosophie, on ne mange pas avec les mains, on n'arrache pas les couverts des mains de son voisin et on n'utilise que les couverts à proximité (celui de gauche et celui de droite). Toute la difficulté du problème (et son intérêt) réside donc dans le choix d'une méthode qui permette de synchroniser les actions des philosophes.

Un philosophe peut être dans l'un des trois états suivants :

- il pense, et dans ce cas il a la politesse de lâcher ses couverts ;
- il est affamé, ce qui l'empêche de penser et le pousse à essayer de saisir les deux couverts qui sont à sa proximité pour pouvoir manger ;
- il mange, et utilise dans ce cas deux couverts.

Un philosophe reste un homme ; il ne peut donc penser pendant un temps indéfini et doit finir par manger. Toutefois, même affamé, un philosophe est un homme parfaitement éduqué et ne mange pas tout son plat d'un seul coup. Nous modéliserons ces deux aspects en considérant d'une part qu'un philosophe reste dans un des deux états *pensant* ou *mangeant* pendant une durée aléatoire et, d'autre part, en imposant un nombre fixe de fois où il doit manger.

Modélisation

Il nous faudra donc recourir à trois classes pour modéliser cette situation : une classe `Philosophe` dont les instances seront les philosophes que nous avons évoqués, une classe `Fourchette` dont les instances seront les fourchettes qu'ils utilisent (en nombre égal au nombre de philosophes) et une classe `TableDePhilosophes`, qui contiendra l'ensemble des éléments (fourchettes et philosophes) impliqués dans le traitement.

Nous définirons `TableDePhilosophes` comme une sous-classe de `Object` (les trois variables d'instance – `nombreDeDineurs`, `fourchettes`, `philosophes` – parlent d'elles-mêmes). La définition de la classe `Philosophe` comprend les variables `nom` (l'identifiant du philosophe), `etat` indique s'il pense, mange ou est affamé, tandis que `fourchetteGauche` et `fourchetteDroite` indiquent les numéros des fourchettes qui se trouvent respectivement à la gauche et à la droite de chaque philosophe.

```
Object subclass: #TableDePhilosophes
     instanceVariableNames: 'nombreDeDineurs fourchettes philosophes '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Philosophes'
```

```
Object subclass: #Philosophe
     instanceVariableNames: 'fourchetteGauche fourchetteDroite nom etat '
     classVariableNames: ''
     poolDictionaries: ''
     category: Philosophes
```

Les couverts (tous identiques et représentés ici par des fourchettes) sont des ressources rares sur lesquelles les philosophes font des accès concurrents. Ils peuvent donc être modélisés par des sémaphores d'exclusion mutuelle, garantissant ainsi l'usage d'une ressource (une fourchette) par un seul processus à la fois (un philosophe). La classe `Fourchette` est donc définie essentiellement avec la variable d'instance `semaphore`.

```
Object subclass: #Fourchette
     instanceVariableNames: 'semaphore '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Philosophes'
```

Gestion des accès concurrents

L'initialisation d'une fourchette doit créer le sémaphore qui la protège contre les accès concurrents :

```
initialize
     semaphore := Semaphore forMutualExclusion
```

On accédera donc à une fourchette en envoyant le message `wait` à son sémaphore, et la fourchette sera libérée par l'envoi du message `signal` à son sémaphore. Ainsi les méthodes `wait` et `signal` définies sur `Fourchette` se contentent-elles de retransmettre ces messages au sémaphore :

```
signal
     semaphore signal.
wait
     semaphore wait.
```

La méthode d'initialisation d'une `TableDePhilosophes` crée les fourchettes et les philosophes (lignes 5 à 7) – le « nom » d'un philosophe est son numéro – et répartit les fourchettes aux philosophes (lignes 8 à 11, les fourchettes sont associées aux philosophes dans l'ordre de placement autour de la table) :

```
 initialize: n
  self nombreDeDineurs: n.
  self fourchettes: (Array new: n).
  self philosophes: (Array new: n).
  1 to: n do: [:p |
    self fourchettes at: p put: (Fourchette new).
    self philosophes at: p put: (Philosophe new: p)].
  1 to: n do: [:p |
    (self philosophes at: p)
     fourchetteGauche: (self fourchettes at: p)
  fourchetteDroite: (self fourchettes at: ((p\\n) + 1))].
  "(p\\n)+1 est egal à p + 1, sauf pour p=n où il vaut alors 1"
```

La méthode de lancement du dîner (méthode d'instance de `TablesDePhilosophes`) est alors définie de cette façon :

```
dine: rep
   1 to: self nombreDeDineurs do: [:p |
     (self philosophes at: p) philosopheCycle: rep].
```

Dans cette dernière méthode, `rep` représente le nombre de cycles (penser, récupérer des couverts, manger, relâcher ses couverts...) que doit effectuer un philosophe.

Cycle de fonctionnement d'un philosophe

Puisque les comportements des philosophes sont autonomes, nous pouvons les modéliser par des processus concurrents, chaque processus étant une suite de changements d'états. Le code présenté ci-après définit un processus par l'envoi du message `fork` à un bloc qui fait passer un philosophe par les actions successives *penser*, *récupérer deux fourchettes*, *manger*, *relâcher ses fourchettes* :

```
philosopheCycle: rep
     [rep timesRepeat: [
        self pense.
        self recupereFourchettes.
        self mange.
        self relacheFourchettes].
     self etat: #philosopheEndormi.
```

```
    ] fork
```

Pour introduire un peu de variabilité dans les comportements des philosophes, nous introduisons une durée aléatoire (classe `Random`) dans les méthodes `mange` et `pense` :

```
mange
      self etat: #philosopheMangeant.
      (Random new  next * 20) rounded  timesRepeat: [Processor yield]
```

```
pense
      self etat: #philosophePensant.
      (Random new  next * 20) rounded timesRepeat: [Processor yield]
```

L'utilisation de `Processor yield` permet de laisser la main aux autres processus lorsqu'un processus (un philosophe) est en train de manger ou de penser. Pour passer de l'état « pensant » à l'état « mangeant », un philosophe doit saisir deux couverts, ce qui correspond techniquement à la réservation sur le sémaphore de ces ressources rares (`fourchetteGauche wait` ; `fourchetteDroite wait`), et les relâcher (`fourchetteGauche signal` ; `fourchetteDroite signal`) dès qu'il s'arrête de manger.

Gestion des problèmes d'inter-blocage

On résout bien ainsi le problème de la *synchronisation*, mais pas celui de l'*inter-blocage*. Supposons en effet que tous les philosophes soient affamés en même temps, et saisissent tous, par exemple, la fourchette qui est à leur droite. Aucun ne pourra accéder à la fourchette du côté opposé, puisque c'est son voisin qui la tient. Il attendra donc que son voisin s'arrête de manger et qu'il repose son couvert. Le problème est que ce voisin est lui-même bloqué en attente de son autre voisin, lui-même bloqué… À moins qu'un des philosophes ne se rende compte du problème et ne se sacrifie en reposant sa fourchette (ce qui ne peut pas se produire car chacun sait qu'un philosophe affamé n'est pas capable de penser), la situation est irrémédiablement bloquée.

Parmi toutes les solutions qui ont été proposées pour résoudre ces problèmes d'inter-blocage, il en est une qui s'applique facilement au cas des philosophes : il suffit de créer un comportement asymétrique chez les philosophes. Un philosophe assis à une place paire saisira toujours en premier la fourchette qui est à sa droite, tandis qu'un philosophe assis à une place impaire saisira celle de gauche (ligne 3). Nous pouvons ainsi écrire les méthodes `recupereFourchettes` et `relacheFourchettes` de la façon suivante :

```
  recupereFourchettes
      self etat: #philosopheAffame.
      nom \\ 2 == 0
        ifTrue:
           [self fourchetteGauche wait.
           self fourchetteDroite wait.]
        ifFalse:
           [self fourchetteDroite wait.
           self fourchetteGauche wait.]
```

```
relacheFourchettes
      self etat: #philosophePosant.
      self fourchetteDroite signal.
      self fourchetteGauche signal.
```

Les méthodes que nous venons de définir permettent (si on y ajoute les méthodes d'accès et d'affectation habituelles, qui n'ont pas été reproduites ici) d'effectuer une simulation d'un dîner de philosophes.

Exemples de fonctionnement

Si nous souhaitons avoir une idée de ce qui se passe effectivement, nous pouvons modifier la méthode `philosopheCycle:` pour demander des affichages dans le `Transcript` :

```
philosopheCycle: rep
     [rep timesRepeat: [
        self pense.
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état
#philosophePensant '.
        self recupereFourchettes.
        self mange.
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état
#philosopheMangeant '.
        self relacheFourchettes
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état #
philosophePosant '.].
     self etat: # philosophePensant.
     Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état #
philosophePensant '.$$$$pourquoi ce style « a revoir ?rectifier au besoin$$$
] fork
```

On peut ainsi obtenir un affichage qui permet de retracer l'évolution des événements :

```
Philosophe 1 est dans l'état #philosophePensant
Philosophe 2 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosophePensant
Philosophe 4 est dans l'état #philosophePensant
Philosophe 5 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosopheAffame
Philosophe 3 est dans l'état #philosopheMangeant
Philosophe 1 est dans l'état #philosopheAffame
Philosophe 1 est dans l'état #philosopheMangeant
Philosophe 3 est dans l'état #philosophePosant
Philosophe 3 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosopheAffame
Philosophe 3 est dans l'état #philosopheMangeant
Philosophe 4 est dans l'état #philosopheAffame
Philosophe 2 est dans l'état #philosopheAffame
Philosophe 5 est dans l'état #philosopheAffame
Philosophe 1 est dans l'état #philosophePosant
Philosophe 1 est dans l'état #philosophePensant
Philosophe 5 est dans l'état #philosopheMangeant
Etc.
```

$$$$En résumé ? ? ?$$$