UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 4

Programmation parallèle

Gabriel Girard[1]

Sherbrooke

26 janvier 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 4

Programmation Parallèle de haut niveau

Ce chapitre est inspiré de [6, 9, 10].

Historiquement, les systèmes parallèles (incluant les systèmes d'exploitation) furent développés en langages de bas niveau. Les arguments avancés contre l'usage des langages évolués pour concevoir ces logiciels étaient généralement ceux-ci :

1. ils ne fournissaient aucun mécanisme pour développer du code adaptée à une architecture particulière (ex. pilote de périphérique) ;
2. ils ne fournissaient aucun outil approprié pour concevoir des programmes parallèles ;
3. ils ne généraient pas du code efficace.

Tous ces arguments ont été effacés par le développement de langages qui supportent la concurrence et génèrent du code extrêmement efficace. La plupart des langages modernes sont très bien pourvus pour supporter la programmation parallèle et offrent des primitives ou des bibliothèques afin de manipuler les fils et les outils de synchronisation. Toutefois, peu d'entre eux fournissent des fonctionnalités adéquates permettant de manipuler des processus.

Nombreux aussi sont les interpréteurs de langages modernes aptes à générer du code très performant, en particulier les langages compilés comme C++. Les compilateurs pour ces langages offrent généralement plusieurs niveaux d'optimisation. Le plus bas niveau, appelé souvent « *quick-and-dirty* », traduit très rapidement mais génère du code très peu efficace et parfois volumineux. C'est généralement le niveau par défaut. Il est couramment choisi dans les institutions d'enseignement ou dans les milieux de développement où la priorité est de générer rapidement plutôt que d'optimiser la performance du code généré. Lors de la compilation d'une version finale pour la production, un niveau d'optimisation assez élevé devrait être privilégié afin de générer du code efficace et cela, en dépit d'un temps de traitement significativement plus long. En fait, en comparant le code généré par certains interpréteurs, on constaterait sans doute que celui-ci est mieux optimisé que ce qu'on aurait pu écrire soi-même.

Notons que dans certains cas, les optimisations impliquent le ré-ordonnancement de certaines instructions, l'élimination d'instructions, de multiples améliorations sur la gestion des données en mémoire, etc [1].

1. Pour plus d'information sur ce sujet, consultez les pages web suivantes [3, 14, 12, 2, 5]

Enfin,il faut savoir que certains langages permettent d'accéder directement aux fonctionnalités du matériel (ou du moins au langage d'assemblage).

En résumé, les langages évolués présentent les avantages d'être plus faciles à tester, à vérifier, à modifier et à transporter. Ceux-ci contribuent à améliorer de façon significative le temps de développement.

4.1 Modèles de haut niveau ou théoriques pour le parallélisme

Les langages utilisés pour écrire des programmes parallèles se doivent de fournir des facilités pour représenter la concurrence, programmer de façon modulaire en plus de proposer des outils de communication et de synchronisation (si requis). Les sections suivantes présentent quelques modèles développés pour spécifier ou implanter le parallélisme.

4.1.1 Les réseaux de Petri

Un réseau de Petri est un modèle mathématique décrivant les calculs parallèles à un degré élevé de détails. Il est constitué d'outils graphiques et mathématiques permettant de modéliser et de vérifier le comportement dynamique des systèmes à événements discrets (informatiques, industriels...). Ainsi, le problème du dîner des philosophes peut être modélisé avec un réseau de Petri.

En fait, les diagrammes d'activités `UML` sont des dérivés simplifiés des réseaux de Petri auxquels on a soustrait la représentation mathématique qui autorisait les preuves formelles.

Pour les systèmes de grandes tailles et complexes, ce modèle peut mener à des descriptions très complexes et de grandes dimensions.

4.1.2 Les modèles à flots de données

Ce modèle théorique a servi à modéliser les idées de calculs parallèles au niveau de l'application. Selon ce modèle, un programme est représenté par un graphe orienté dont les nœuds correspondent aux énoncés et les arcs, aux données qui circulent d'une instruction à l'autre, implantant ainsi le principe de «flots de données». Notons que les langages à flots de données et les langages fonctionnels partagent plusieurs principes.

Plusieurs langages, supportant le concept de flots de données, ont été développé tels Linda, Lucid, Simulink, Verilog.

4.1.3 Le modèle des «acteurs»

Le modèle basé sur les «acteurs» est un modèle mathématique, proposé par Carl Hewitt au début des années 1970, pour manipuler des preuves formelles et certains modèles spécifiques au robot. La notion d'acteurs ressemble à celle de l'objet, soit une entité définie par quelques caractéristiques. On considère les acteurs comme les seules fonctions primitives nécessaires à la programmation concurrente. Les acteurs communiquent par envois/réceptions de messages. En répondant à un message, un acteur peut effectuer un calcul, instancier d'autres acteurs ou envoyer d'autres messages.

Les langages Erlang et Scala supportent ce modèle alors que les langages SR et JR ne le font que partiellement grâce aux processus et au passage de messages.

4.1.4 Le modèle objet

Le modèle objet est très bien connu. La plupart des mécanismes que nous présentons (et des langages que nous utilisons) sont basés sur celui-ci.

4.1.5 Processus et fils d'exécution

Les processus et fils d'exécution sont les principales unités auxquelles on a recours lors de la conception de programmes parallèles. Les langages de haut niveau doivent fournir la capacité de créer des processus (ou fils d'exécution), de les synchroniser et de les faire communiquer.

Au niveau des processus (ou fils), il y a deux moyens de communiquer : par variables communes ou par messages. Lorsque les processus communiquent par variables communes, la personne en charge du développement de l'application doit spécifier elle-même la synchronisation. Dans le cas des messages, la synchronisation est implicite (faites par le système sous-jacent).

Un processus est une forme d'objet en ce sens qu'il possède des variables locales et des fonctions (programme séquentiel). Les données locales ne peuvent être accédées qu'en exécutant le code de ce processus. En général, aucun accès direct n'est permis. S'il y a des données globales, celles-ci sont normalement englobées par des procédures ou intégrées dans des types abstraits de données.

4.1.6 Procédures et fonctions

Le concept de fonctions (routines, procédures ou méthodes), aussi très généralisé, permet de diviser un programme en de multiples modules. Ce sont fréquemment ces unités qui servent de base aux fils d'exécution.

4.1.7 Type abstrait de données

Les procédures sont relativement limitées. Pour permettre de vraiment encapsuler les données, un mécanisme plus complet a été développé : les types abstraits de données. Cette approche favorise la création de nouveaux types possédant leurs propres opérations. Les types abstraits de données sont généralement présents dans tous les langages sous forme de «`class`» ou autres mots clés.

4.2 Région critique et région critique conditionnelle

Il est possible que les différentes opérations sur une même variable s'exécutent simultanément. Comme nous l'avons déjà constaté, cette situation est une source potentielle d'erreurs. La synchronisation de ces opérations est donc primordiale. Les sémaphores (présentés au chapitre précédent) s'avèrent plutôt des outils de bas niveau (tel l'assembleur). De nouveaux outils de synchronisation plus «évolués» sont requis.

4.2.1 Les régions critiques

Comme première solution, Hoare et Brinch Hansen ont créé une notation structurée pour spécifier la synchronisation, les «régions critiques».

Au fil des années, deux notations ont été employées pour les décrire :

Per Brinch Hansen : «IEEE Computer Pioneer Award»

Per Brinch Hansen [15] est reconnu pour ses contributions dans le domaine des systèmes d'exploitation, de la programmation concurrente et des systèmes parallèles et distribués. En particulier, il est reconnu pour être à l'origine des concepts suivants :

- le noyau de système d'exploitation (RC4000). C'est en fait le premier micro-noyau ;
- la séparation des politiques et des mécanismes dans les systèmes ;
- les moniteurs ;
- l'appel de procédure éloigné (RPC).

Il a aussi conçu plusieurs langages de programmation parallèle (Concurrent Pascal, Edison, Distributed Process, ...).
Il a obtenu en 2002 le prix «IEEE Computer Pioneer Award», pour ses travaux dans le domaine des systèmes d'exploitation et de la programmation concurrente.

1. La notation de Peterson
   Soit une variable v de type T partagée par plusieurs processus, celle-ci est déclarée de la façon suivante :
   var v : **shared** T;
   La variable v ne peut alors être accédée qu'à l'intérieur d'un énoncé «`region`» tel que :
   region v **do** S;
   Cela signifie que l'accès à la variable v par l'énoncé S se fera en exclusion mutuelle.
2. La notation générale
   Une notation plus générale permet d'inclure plusieurs variables dans une même ressource partagée de la façon suivante :
   var $v_1, v_2, ..., v_n : T$;
   ressource $R : v_1, v_2, ..., v_n$;
   L'accès se fait toujours à l'intérieur d'un énoncé «`region`» de la façon suivante :
   region R **do** S;

Cette construction signifie que pendant l'exécution de S, aucun autre processus ne peut accéder à la ressource R (ou la variable v). L'exclusion mutuelle est assurée. Ainsi, deux processus exécutant simultanément

region R **do** $S1$;
region R **do** $S2$;

produiront toujours un résultat équivalent à «$S1; S2$» ou «$S2; S1$».

Il y a peu de différence entre ces deux notations outre le fait que la seconde permet de couvrir plusieurs variables.

Les régions critiques ressemblent aux variables atomiques de C++.

Les région critiques permettent d'éviter les erreurs simples causées par le recours aux sémaphores (oublier un P ou un V, etc).

Implantation

Le compilateur en charge de traduire le langage qui fournit les régions critiques, doit générer le code nécessaire pour assurer l'exclusion mutuelle. Ce code de bas niveau utilise évidemment les sémaphores pour assurer la synchronisation. Ainsi pour chaque ressource R (ou variable v), le compilateur génère un sémaphore distinct pour assurer l'exclusion mutuelle (initialisé à 1). Le programme 4.1 présente le code produit par le compilateur. L'exclusion mutuelle est alors garantie.

```
1        region R do S;
2
3  Génère
4
5        semaphore mutex = 1;
6        ...
7        P(mutex);
8        S;
9        V(mutex);
```

Programme 4.1 – Implantation des régions critiques

À noter que l'imbrication d'énoncés de type «régions critiques» est possible (un énoncé région contenu dans un autre énoncé région). Comme on le constate au programme 4.2, une utilisation inadéquate des régions imbriquées risque de mener à un interblocage. Clairement, l'exécution simultanée de $P1$ et $P2$ peut provoquer une étreinte fatale (autre nom de l'interblocage). Pour éviter cette situation, on a recourt à une technique de traitement des interblocages. Ici, la plus simple consiste à imposer un ordre dans l'allocation des ressources.

```
1    ressource X:a; Y:b;
2    ...
3    parbegin
4       P1: region X do region Y do S0;
5       P2: region Y do region X do S1;
6    parend
```

Programme 4.2 – Régions critiques imbriquées

4.2.2 Les régions critiques conditionnelles

Les régions critiques, telles que décrites précédemment, ne permettent pas d'exprimer la synchronisation conditionnelle. Pour y arriver, on leur ajoute un énoncé **when** qui les transforme en **région critique conditionnelle** (RCC).

Voici la syntaxe des régions critiques conditionnelles :

var $v_1, v_2, ..., v_n$: T;
ressource R : $v_1, v_2, ..., v_n$;
...
region R **when** B **do** S;

où B est une expression booléenne. Ainsi, un processus sera bloqué sur la condition B jusqu'à ce qu'elle soit vraie. Cette notation, en plus de garantir l'exclusion mutuelle, assure aussi la synchronisation conditionnelle.

Quelques éléments importants à connaître pour le bon fonctionnement de cet outil :

- Afin de garantir que B restera vraie pendant toute l'exécution de S, l'évaluation de B et l'exécution de S doivent être ininterruptibles.
- Si la condition B est fausse, alors en plus de se bloquer, le processus doit relâcher l'accès exclusif à la région.
- Le mécanisme de délai doit être équitable, i.e. si un processus est en attente sur une condition B, qui prend de façon répétitive la valeur vraie, alors le processus devra éventuellement être débloqué.

Le programme 4.3 présente une implantation de notre mini système d'exploitation par lots en utilisant les régions critiques conditionnelles. Il est intéressant de noter dans cet exemple la séparation claire entre la synchronisation conditionnelle et l'exclusion mutuelle. En premier lieu, l'exclusion mutuelle est implicite dans un énoncé `region`. Puis, les expressions booléennes, dans les énoncés RCC qui contrôlent les accès aux tampons, spécifient clairement les conditions requises pour autoriser cet accès.

Implantation

Un compilateur, traduisant un langage qui fournit les RCC, doit pouvoir générer des instructions primitives pour chacun de ces énoncés. Ces «instructions primitives» incluent évidemment l'usage de sémaphores.

Le compilateur associe donc à chaque ressource R les variables suivantes :

- Un sémaphore «`R_mutex`» pour assurer l'exclusion mutuelle (initialisé à 1) ;
- Un sémaphore «`R_wait`» pour assurer la synchronisation conditionnelle (attente si B est faux) (initialisé à 0) ;
- Un entier «`R_count`» pour compter le nombre de processus en attente sur «`R_wait`» (initialisé à 0) ;
- Un entier «`R_temp`» pour compter le nombre de processus ayant re-tester la condition» (initialisé à 0) ;

La présence de cette dernière variable est due au fait que chaque processus quittant la section critique peut changer la condition booléenne, donnant ainsi la chance à un autre processus d'entrer en section critique. Lorsque ceci se produit, tous les processus en attente sur cette condition (R_wait) doivent avoir la chance de re-tester leur condition booléenne afin de vérifier si l'accès leur est permis. La variable «`R_temp`» sert donc à déterminer si tous les processus ont réévalué leur condition.

Le programme 4.4 présente le code généré par un compilateur pour l'énoncé :

region R **when** B **do** S;

Selon ce code :

- la ligne 7 sert à assurer l'exclusion mutuelle sur tous les énoncés de la région ;
- la ligne 8 teste l'expression booléenne ;
- les lignes 10 à 21 présentent les actions effectuées lorsque la condition est fausse :
  1. on incrément le nombre de processus bloqué (ligne 10) ;
  2. on libère l'exclusion mutuelle (ligne 11) ;

```
1  Program OPSYS;
2  type      tampon(T): record
3                          item : array[0..n-1] of T;
4                          tete, queue : 0.. n-1 initial (0,0);
5                          dimension : 0..n initial (0);
6                        end;
7  var       tampon_in : tampon(entree);
8            tampon_out : tampon(sortie);
9
10 resource  ib : tampon_in;
11           ob : tampon_out;
12
13 process lecteur;
14    var ligne : entree;
15    loop
16       ...lecture ligne;
17       region ib when tampon_in.dimension < n do;
18            tampon_in.item[tampon_in.queue] := ligne;
19            tampon_in.dimension := tampon_in.dimension + 1;
20            tampon_in.queue := (tampon_in.queue + 1) mod n;
21       end;
22    end;
23 end process;
24
25 process traitement;
26    var ligne : entree;
27        resultat : sortie;
28    loop
29       region ib when tampon_in.dimension > 0 do;
30            ligne := tampon_in.item[tampon_in.tete];
31            tampon_in.dimension := tampon_in.dimension - 1;
32            tampon_in.tete := (tampon_in.tete + 1) mod n ;
33       end;
34       ..... traitement de ligne et génération de resultat;
35       region ob when tampon_out.dimension < n do;
36            tampon_out.item[tampon_out.queue] := resultat;
37            tampon_out.dimension := tampon_out.dimension + 1;
38            tampon_out.queue := (tampon_out.queue + 1) mod n;
39       end;
40    end;
41 end process;
42
43 process imprimante;
44    var resultat : sortie;
45    loop
46       region ob when tampon_out.dimension > 0 do;
47            resultat := tampon_out.item[tampon_out.tete];
48            tampon_out.dimension := tampon_out.dimension - 1 ;
49            tampon_out.tete := (tampon_out.tete + 1) mod n ;
50       end;
51       ... impression de resultat;
52    end;
53 end process;
```

Programme 4.3 – Système d'exploitation par lots

3. on se bloque (ligne 12) ;
4. le contrôle revient ! Il faut tester de nouveau la condition booléenne (ligne 13) ;
5. B est toujours faux, on regarde si un autre processus bloqué peut tester sa condition (ligne 15-16) ;
6. si un autre processus peut valider sa condition, on le libère (ligne 17) ;
7. aucun processus ne peut tester de nouveau sa condition, on libère l'exclusion mutuelle (ligne 18) ;
8. on se bloque de nouveau (ligne 19) ;
9. B est vrai (ligne 13), on passe en section critique (ligne 21 et 22)

```
1  var R_mutex, R_wait : semaphore;
2      R_count, R_temp : integer};
3
4  R_count := R_temp := R_wait := 0 ;
5  R_mutex := 1;
6
7  P(R_mutex);
8  if not B then
9  begin
10     R_count := R_count + 1;
11     V(R_mutex);
12     P(R_wait);
13     while not B do
14     begin
15         R_temp := R_temp + 1;
16         if R_temp <  R_count
17            then V(R_wait);
18            else V(R_mutex);
19         P(R_wait);
20     end;
21     R_count := R_count -1;
22 end;
23 ......Section critique.....
24 if R_count > 0
25 then
26    begin
27       R_temp := 0;
28       V(R_wait);
29    end;
30 else V(R_mutex);
```

Programme 4.4 – Implantation des RCC

Améliorations

Cette première version des régions critiques conditionnelles ne permet d'attendre sur une condition qu'en tout début de la section. Il existe cependant des situations où la synchronisation doit se situer ailleurs qu'au début, et c'est pour tenir compte de cette autre situation que Brinch Hansen a proposé une version modifiée des RCC. Celle-ci permet de tester la condition n'importe où dans la région. Cette nouvelle construction possède la structure suivante :

```
region R do
begin
    S1;
    await(B);
    S2;
end
```

L'attente conditionnelle est ainsi réalisable entre deux énoncés quelconques d'une région critique. Le programme 4.5 présente une solution au problème des lecteurs/écrivains basée sur cette nouvelle construction. Celle-ci combine l'utilisation d'une classe et des RCCs. Quant à lui, le programme 4.6 illustre l'emploi de cette classe faite par des lecteurs et des écrivains.

```
1  type reader-writer = class
2       var  v : shared record
3                    nreaders, nwriters : integer;
4                    busy : boolean;
5                end;
6
7    procedure entry open-read;
8       region v do};
9           await(nwriters = 0);
10          nreaders := nreaders + 1 ;
11      end;
12   end procedure;
13
14   procedure entry close-read;
15      region v do;
16          nreaders := nreaders - 1 ;
17      end;
18    end procedure;
19
20   procedure entry open-write;
21      region v do;
22          nwriters := nwriters + 1;
23          await((not busy) and (nreaders = 0));
24          busy := vrai;
25     end;
26   end procedure;
27
28   procedure entry close-write;
29      region v  do;
30          nwriters := nwriters - 1;
31          busy := faux;
32      end;
33    end procedure;
```

Programme 4.5 – Implantation d'une classe pour les lecteurs/écrivains avec des RCCs

Cet exemple comporte cependant un défaut important. Le concept de classe, tel que défini, ne garantit pas que tous les lecteurs et écrivains obéiront aux règles. Ainsi, certains peuvent «oublier» d'obéir aux règles en omettant de faire un «open_read», un «open_write», un «close_read» ou un «close_write». Notons que ce comportement est similaire à celui rencontré lorsqu'on a recourt aux sémaphores.

```
1  var rw: reader_writer;
2
3  // Lecteur ....
4      rw.open_read()
5        lecture
6      rw.close_read()
7
8  // Écrivain ....
9      rw.open_write()
10       lecture
11     rw.close_write()
```

Programme 4.6 – Utilisation de la classe lecteurs/écrivains

Une solution est proposée en ce sens au programme 4.7. Ce dernier intègre les lectures et les écritures dans la classe elle-même afin de forcer le respect des règles. Les fonctions de lecture et d'écriture deviendraient locales (privées) à la classe et elles seules accéderaient à la ressource. Cette approche garantie un accès correct et assure le respect des règles. Il est important de constater que le problème a été résolu par une redéfinition de la classe et, non pas, par un changement dans la synchronisation.

Mais...Cette solution n'est toujours pas satisfaisante ! En effet, elle limite un processus particulier à un seul accès à la fois. Un processus pourrait effectivement nécessiter de multiples accès comme :

```
    // Lecteur ....
        rw.open_read()
            lecture
            lecture
            lecture
        rw.close_read()
```

La solution à ce besoin particulier consisterait alors à redéfinir la classe. Toutefois, cela suppose une connaissance à priori de l'utilisation de la ressource. Cette condition s'avère délicate à satisfaire.

Évaluation

Plusieurs critères sont généralement appliqués pour évaluer un outil de synchronisation, soit celui des coûts, celui de la modularité de la construction, celui de la puissance d'expression et celui de la facilité d'utilisation.

Les RCCs se révèlent fort dispendieuses à implanter. Diverses raisons expliquent ces coûts élevés, notamment celle due à l'évaluation de la condition B (du «`when`»). En effet, celle-ci peut contenir des variables locales au processus nécessitant alors un changement de contexte afin de permettre au processus d'évaluer sa condition. De plus, rappelons qu'à chaque changement de la condition, chacun des processus doit pouvoir la ré-évaluer, engendrant ainsi sur un système mono-processeur multi-programmé, un très grand nombre de changements de contexte, et donc, entraînant des pertes coûteuses en temps UCT. Certaines optimisations existent cependant pour réduire celles-ci. Ainsi, sur un système multi-processeurs, les RCCs s'implantent efficacement si chacun des processus réside sur un processeur distinct et s'il procède par attente active.

En ce qui concerne notre deuxième critère d'évaluation, la modularité des RCCs, nous dirons qu'elle aussi laisse à désirer. De fait, les énoncés de type RCC sur une ressource particulière sont

```
1  type reader-writer = class
2       var  v : shared record
3                    nreaders, nwriters : integer;
4                    busy : boolean;
5                end;
6
7    procedure open-read;
8       region v do};
9           await(nwriters = 0);
10          nreaders := nreaders + 1 ;
11      end;
12   end procedure;
13
14   procedure close-read;
15      region v do;
16          nreaders := nreaders - 1 ;
17      end;
18    end procedure;
19
20   procedure open-write;
21      region v do;
22          nwriters := nwriters + 1;
23          await((not busy) and (nreaders = 0));
24          busy := vrai;
25      end;
26   end procedure;
27
28   procedure close-write;
29      region v  do;
30          nwriters := nwriters - 1;
31          busy := faux;
32      end;
33    end procedure;
34
35    procedure entry read(.......);
36           open-read;
37           ... lecture du fichier ...
38           close-read
39    end procedure;
40
41    procedure entry write(........);
42           open-write;
43           ... écriture dans le fichier...
44           close-write;
45   end procedure;;
```

Programme 4.7 – Implantation d'une classe pour les lecteurs/écrivains avec des RCCs

généralement dispersés dans tout le programme. Même s'il est possible de les regrouper dans une classe (comme cela est fait au programme 4.7), cela n'est aucunement une obligation. Le fait est que, même si les RCCs résident majoritairement dans une classe, il est toujours possible de les ajouter directement dans le programme en ignorant la classe. Conclusion ! Quiconque désirant analyser le programme se doit d'examiner tout le code afin de déterminer l'entièreté des usages de la ressource.

Les deux derniers critères, la puissance d'expression et la facilité d'utilisation, sont tous deux à l'avantage des RCCs. En effet, ceux-ci expriment toutes les formes de synchronisations et cela avec une grande facilité.

Même combinée à la notion de classe, chacune des fonctions ou procédures de la classe se doit d'assurer explicitement la synchronisation.

Certains langages, tel Edison, supportaient cette construction. Le langage SR la supporte également.

4.3 Les moniteurs

Afin de résoudre certaines problématiques reliées aux RCCs, Hoare et Brinch Hansen ont développé les sémaphores.

Charles Anthony Hoare : Prix Turing 1980

Charles Antony Richard [13] Hoare est connu pour

- avoir inventé le «quicksort» en 1960 (encore très utilisé aujourd'hui) ;
- avoir écrit le premier compilateur complet pour le langage Algol 60 (l'ancêtre de Pascal et plusieurs autres dont C)
- être à l'origine de la logique de Hoare qui sert à la vérification de programmes ;
- être à l'origine du langage formel Communicating sequential processes (CSP) qui permet de spécifier l'interaction de processus concurrents (y compris le fameux problème du dîner des philosophes) ;
- avoir introduit le concept de moniteur ;
- avoir introduit la spécification formelle de langages de programmation.

Il a obtenu le prix Turing [a] pour « ses contributions fondamentales à la définition et la conception des langages de programmation ».
En 2009, Hoare s'est excusé d'avoir inventé le pointeur NULL en ces termes : « Je l'appelle mon erreur à un milliard de dollars.... »

a. Le prix Turing est un peu l'équivalent du prix Nobel mais en informatique.

Un moniteur est formé en encapsulant la définition de la ressource avec ses opérations. Cette façon de faire est similaire à un type abstrait de données. En fait un moniteur est défini comme une classe pour laquelle on remplace le terme «`class`» par celui de «`monitor`». Ce concept permet à une ressource d'être considérée tel un module.

Le programme 4.8 présente la structure d'un moniteur. Tout comme pour une classe, il contient :

- des variables permanentes ;
- des fonctions pour implanter les opérations sur la ressource. Les fonctions contiennent des

variables locales et peuvent être paramétrisées ;

- du code pour initialiser les variables permanentes. Ce code est exécuté une seule fois, et ce avant toutes autres exécutions.

```
1  Nom-moniteur = monitor
2  var  ... :  ...  // déclaration des variables permanentes;
3
4  procedure op1(param\a`etres);
5     var ... :  ... // déclaration des variables locales à {\bf op1}};
6     begin
7       ... // code pour implanter  op1;
8     end;
9
10    ...
11
12 procedure opN(paramètres);
13    var ... : ... // déclaration des variables locales à  opN
14    begin
15      ... // code pour implanter opN;
16    end;
17
18 begin
19    ... // code pour initialiser les variables permanentes;
20 end;
```

Programme 4.8 – Définition d'un moniteur

Un appel à une opération d'un moniteur se fait de la façon suivante :

`call` *nom_moniteur*`.op`$_i$`(paramètres)`

La caractéristique la plus importante d'un moniteur est que chacune des fonctions ou procédures, à l'intérieur du même moniteur, s'exécute toujours en exclusion mutuelle. Cela se fait automatiquement grâce à la définition du moniteur. La personne qui développe n'a pas à s'en préoccuper.

Um moniteur résout donc l'enjeu de l'exclusion mutuelle, celle-ci devenant implicite. Pour assurer la synchronisation conditionnelle, plusieurs constructions ont été proposées et sont présentées dans les sections suivantes.

4.3.1 Les variables de type condition avec les opérations wait/signal

Pour assurer la synchronisation conditionnelle, la première construction proposée a été celle des variables de type condition, telles que définies par Hoare. Une variable condition doit obligatoirement être déclarée à l'intérieur d'un moniteur et se manipule avec seulement deux opérations, soient `wait` et `signal`. Le programme 4.9 expose la syntaxe de la déclaration d'une telle variable ainsi que son utilisation.

L'exécution du `cond1.wait` bloque le processus appelant et relâche l'exclusion mutuelle sur le moniteur. L'exécution d'un `cond1.signal` fonctionne de la façon suivante :

- Si aucun processus n'est bloqué sur la variable `cond1`, l'appelant continue son exécution et l'opération n'a aucun effet, i.e. la variable condition ne change pas d'état et le signal est perdu.

```
var cond1 : condition;
...
cond1.wait;

cond1.signal;
```

Programme 4.9 – Définition et utilisation des variables de type condition

Attention :

Les variables conditions, contrairement aux sémaphores, n'emmagasinent pas les signaux.

- S'il y a des processus en attente sur la variable, alors le signal réactive un seul processus et bloque l'émetteur du signal pour donner le contrôle au processus réactivé.

  Le processus émetteur du signal poursuivra son exécution seulement lorsqu'il n'y aura plus de processus dans le moniteur. Comme la priorité est donnée aux émetteurs de signaux sur les processus qui tentent d'entrer dans le moniteur, ils compléteront leur exécution ultérieurement.

  Le processus ayant émis le signal se bloque au profit du processus nouvellement débloqué car si n'était pas le cas, la condition pourrait ne plus être vraie lorsque le processus débloqué reprendrait le contrôle. En effet, plusieurs événements pouvant affecter le moniteur pourraient se produire entre le moment où le signal est émis et le moment où le processus reprend le contrôle. Ce transfert de contrôle immédiat assure qu'aucun événement affectant le moniteur ne se produise avant que le processus signalé ne s'exécute. En fait, l'émetteur du signal transfert «l'exclusion mutuelle» qu'il détient, directement au processus signalé.

Implication au niveau de la programmation

Ce comportement au niveau des opérations `wait` et `signal` amène une implication importante au niveau de la programmation avec des variables conditions. Comme les signaux ne s'accumulent pas, il est possible de les «manquer». Il faut donc combiner l'utilisation des variables de type condition, principalement de l'opération `wait`, avec une «condition» à tester. Son emploi doit donc se faire obligatoirement à l'intérieur d'un énoncé de sélection de la manière suivante :

```
if (cond) .... wait().
```

Le programme 4.10 propose une solution à la gestion des tampons (producteurs/consommateurs) basée sur un moniteur. Une version modifiée de notre petit système d'exploitation par lots qui utilise ce moniteur est fournie au programme 4.12. Le programme 4.11 implante une version modifiée de la fonction `retirer` du moniteur contrôlant l'accès au tampon. Cette solution fonctionne-t-elle ? Si vous affirmez qu'elle ne fonctionne pas, donnez une séquence d'appels aux fonctions `deposer` et `retirer` qui provoquera une erreur ? Quel type d'erreur cela produira-t-il ?

```
1  type tampon(T) = monitor
2  var  item :  array[0..n-1] of T;
3       tete, queue : 0.. n-1;
4       dimension : 0..n;
5       non-plein, non-vide : condition;
6
7  procedure deposer(p : T);
8    begin
9       if dimension = n then non-plein.wait;
10        item[queue] := p;
11        dimension := dimension + 1;
12        queue := (queue + 1) mod n;
13        non-vide.signal;
14   end;
15 procedure retirer(var p : T);
16   begin
17      if dimension = 0 then non-vide.wait;
18      p := item[tete];
19      dimension := dimension - 1;
20      tete := (tete + 1) mod n;
21      non-plein.signal;
22   end;
23 begin
24    dimension := 0; tete := 0; queue := 0;
25 end;
```

Programme 4.10 – Implantation d'un type tampon avec un moniteur

```
1  type tampon(T) = monitor
2  ...
3
4  procedure retirer(var p : T);
5    begin
6       non-vide.wait;
7       p := item[tete];
8       dimension := dimension - 1;
9       tete := (tete + 1) mod n;
10      non-plein.signal;
11   end;
```

Programme 4.11 – Fontion retirer sans le test de la condition

```
1  Program OPSYS;
2  type   tampon(T) =  ... // moniteur précédant
3  var    tampon_in : tampon(entree);
4         tampon_out : tampon(sortie);
5  process lecteur;
6    var ligne : entree;
7    loop
8       lecture ligne;
9       call tampon_in.deposer(ligne);
10   end;
11 end process;
12 process traitement;
13    var  ligne : entree;
14      resultat : sortie;
15    loop
16       call tampon_in.retirer(ligne);
17       ... //traitement de ligne et génération de resultat;
18       call tampon_out.deposer(resultat);
19    end;
20 end process;
21 process imprimante;
22    var resultat : sortie;
23    loop
24       call tampon_out.retirer(resultat);
25       ... // impression de resultat
26    end;
27 end process;
```

Programme 4.12 – Système d'exploitation par lots

La programme 4.13 implante un sémaphore à l'aide des moniteurs. Cette solution ne fournit pas un sémaphore général mais seulement un sémaphore binaire.

```
1  type semaphore = monitor
2
3  var   occupe : boolean;
4        non-occupe : condition;
5
6  procedure P();
7    begin;
8       if busy then non-occupe.wait;
9       occupe := vrai;\\
10   end;
11 procedure V();
12   begin;
13      occupe := faux;
14      non-occupe.signal;
15   end;
16 begin
17    occupe := faux;
18 end;
```

Programme 4.13 – Implantation d'un sémaphore avec les moniteurs

Une solution au problème des philosophes basée sur les moniteurs est implantée par le programme 4.14 et son utilisation par les philosophes est présentée au programme 4.15. Cette solution est quasi fonctionnelle. Elle élimine toute possibilité d'interblocage. Elle garantit cette caractéristique en forçant un philosophe à prendre ses baguettes seulement si toutes deux sont libres. Ainsi un philosophe i peut mettre son état à «mange» seulement si les états de ses voisins immédiats sont différents de «mange». Cette solution introduit cinq variables conditions qui permettent à un philosophe de se bloquer s'il a faim et qu'il ne peut obtenir ses baguettes.

La fonction «`tester`» est la fonction critique dans cette solution. Elle permet à un processus i de tester s'il peut manger. Elle sert aussi lorsqu'un philosophe i cesse de manger. Lorsque cela se produit, ce dernier doit vérifier si ses voisins peuvent manger. Si c'est le cas, il leur envoie un signal. C'est cette fonction qui peut nuire au bon fonctionnement de la solution, et ce non pas dû à un code fautif mais plutôt au fait qu'elle s'exécute aussi en exclusion mutuelle. Le processus tente alors de prendre l'exclusion mutuelle une seconde fois sur le même moniteur. Cette façon de faire peut générer une forme d'interblocage du processus avec lui-même (il attend que l'exclusion mutuelle soit libérée par lui-même). Une solution envisageable consisterait à demander que la fonction «`tester`» ne s'exécute pas en exclusion mutuelle (comme `SR` le fait) ou bien de recourir à des sémaphores dits récursifs pour implanter l'exclusion mutuelle sur le moniteur. Un sémaphore, dit récursif, détecte, lors d'un appel, si le sémaphore est détenu par le processus appelant. Si c'est le cas, il ne bloque pas l'appelant (il ne fait rien).

Implantation

Pour chaque moniteur défini dans un langage, le compilateur associé crée un sémaphore d'exclusion mutuelle (`mutex`) initialisé à 1. La synchronisation conditionnelle exige l'usage de plusieurs variables :

```
1  type philosophes = monitor
2
3  var   etat : array[0..4] of (pense, afaim, mange);
4        attente : array}[0..4] of condition;
5
6  procedure prendre(i : 0..4);
7    begin;
8       etat[i] := afaim;wait
9       tester(i);
10      if etat[i] != mange then} attente[i].wait;
11   end;
12
13 procedure deposer(i : 0..4);
14   begin;
15      etat[i] := pense;
16      tester(i-1 mod 5);
17      tester(i+1 mod 5);
18   end;
19
20 procedure tester(k : 0..4);
21   begin;
22      if (etat[k+1 mod 5] != mange) and
23          (etat[k] = afaim) and (etat[k-1 mod 5] != mange)
24      then begin
25             etat[k] := mange;
26             attente[k].signal;
27           end;
28   end;
29 begin
30    for i:=0 to 4 do etat[i] := pense;
31 end;
```

Programme 4.14 – Solution au problème des philosophes avec les moniteurs

```
1            var dp:philosophes;
2            ...
3            dp.prendre(i);
4            .....mange
5            dp.deposer(i);
```

Programme 4.15 – Utilisation du moniteur par les philosophes

- Un sémaphore `sem (= 0)` pour bloquer les émetteurs de `wait` ;
- Une variable `cpt` pour compter le nombre de processus en attente sur `sem` ;
- Un sémaphore `next (= 0)` pour bloquer les émetteurs de `signal`.
- Une variable `next_count` pour compter le nombre de processus suspendu sur `next`

Dans le code généré par le compilateur, chaque fonction du moniteur est encadrée par le code de la figure 4.16.

```
P(mutex)              // debut SC -> on s'assure de l'exclusion mutuelle
   ...
   code pour OPi
   ...
if (next_count > 0)  // Fin SC -> Est-ce qu'il y a un émetteur de signal bloqué?
    then V(next)     // Oui on en débloque un.
    else V(mutex)    // Non on libère le moniteur
```

Programme 4.16 – Code encadrant chaque fonction du moniteur

L'exclusion mutuelle est alors assurée et, à la sortie du moniteur, la priorité est donnée aux émetteurs de signaux (`V(next)`).

Pour chaque opération sur une variable condition, le compilateur doit générer du code particulier. L'opération `wait` génère le code de la figure 4.17,

```
cpt = cpt+1          // Un processus de plus en attente
if (next_count > 0) // Est-ce qu'il y a des émetteurs de signaux en attente
    then V(next)     // Oui, on en libère un
    else V(mutex)    // Non on libère le moniteur
P(sem)               // On bloque le processus
cpt = cpt - 1        // Un processus de moins bloqué sur la variable condition
```

Programme 4.17 – Code généré pour implanter l'opération `wait`.

Cette implantation compte le nombre de processus en attente sur une variable condition (`cpt`).

L'opération `signal` génère le code du programme 4.18

```
if (cpt > 0)             // Y-a-t-il des pcs bloqué sur la variable condition?
    then begin           // Oui -> on en débloque un
       next_count += 1   // Un émetteur de signal de plus en attente
       V(sem)            // On débloque le processus
       P(next)           // L'émetteur de signal se bloque
       next_count -= 1   // Un émetteur de signal de moins en attente
    end
```

Programme 4.18 – Code généré pour implanter l'opération `signal`.

Plusieurs solutions efficaces existent pour implanter des moniteurs. Cette section présente l'une d'entre elles.

4.3.2 Variables de type queue avec les opérations delay/continue

Contrairement aux variables de type condition, un seul processus à la fois est autorisé à se bloquer sur une variable de type queue. Il n'y a donc pas de file d'attente associée à une variable de type queue.

Pour permettre à plusieurs processus d'attendre sur une même condition, on a recourt à un tableau de variables de type queue dont la gestion sert à ordonnancer les processus et assure l'équité (ou non).

Un processus, désirant attendre sur une condition, se doit d'appeler la fonction `delay` sur une variable de type queue (possiblement un élément d'un tableau). Pour débloquer un processus en attente, il suffit d'appeler la fonction `continue` sur cette même variable. L'exécution du `continue` diffère du `signal` car elle réactive le processus bloqué sans bloquer l'appelant. Elle le fait automatiquement sortir du moniteur (souvent appelé `signal and exit`).

Ce mécanisme s'implante plus facilement et à moindre coût que dans le cas des variables conditions combinées aux opérations `signal/wait`. Les variables de type `queue` sont toutefois beaucoup moins souples et plus complexes à utiliser. Brinch Hansen avait ajouté ce mécanisme au langage Pascal concurrent mais il n'est plus présent dans les langages modernes.

4.3.3 Attente conditionnelle et signal automatique

Certains tracas causés par l'emploi des variables de type `condition` ou de type `queue` ont amené Hoare à introduire un énoncé d'attente conditionnelle. Voici le format de cet énoncé :

`wait(`B`)`

où B est une expression booléenne formée de variables soit locales, soit globales, ou des unes et des autres.

L'exécution de cet énoncé bloque le processus jusqu'à ce que l'expression B soit vraie. La réactivation du processus n'exige aucun signal.

Cet énoncé ressemble à l'énoncé `when` des RCCs et présente les mêmes inconvénients au niveau de la performance. Un allocateur de ressource munie de ce mécanisme est implanté par le programme 4.19.

```
1  type A =  monitor
2  var non-occupe : boolean;
3
4  procedure reserve();
5    begin;
6       wait(non-occupe);
7       non-occupe := faux;
8    end;
9  procedure libere();
10   begin;
11      non-occupe := vrai;
12   end;
13 begin
14    non-occupe := vrai;
15 end;
```

Programme 4.19 – Allocateur de ressource

4.3.4 Variables de type condition avec les opérations wait/notify

Cette nouvelle mouture de la variable condition est très similaire à celle vue précédemment. Seule l'opération `notify` (qui remplace l'opération `signal`) apporte une distinction dans le comportement de ce type de variable.

Recourir à cette variante des variables conditions représente surtout un moyen de relâcher l'exclusion mutuelle sur un moniteur. L'opération `wait` a exactement le même effet qu'avant, elle bloque l'appelant et libère l'exclusion mutuelle sur le moniteur. Toutefois, l'exécution de l'opération `notify` sur une telle variable ne bloque pas l'émetteur et permet ainsi à un processus en attente de terminer son exécution ultérieurement (il est débloqué mais ne peut s'exécuter immédiatement). Notons que l'on réfère souvent à l'opération `notify` par l'expression «`signal & continue`».

Ce mécanisme est répandu dans la majorité des environnements fournissant des variables de type condition tel que SR, Java, Posix (PThread), C++, Python, Solaris, Linux et Windows. SR présente la particularité de fournir tous les types de synchronisation pour une opération `signal` (`signal & wait`, `signal & exit`, `signal & continue`).

Implantation

L'implantation de cette version des variables «condition» est relativement simple. Pour chacun des moniteurs définis dans un langage, le compilateur associé crée un sémaphore d'exclusion mutuelle (`mutex`) initialisé à 1. La synchronisation conditionnelle est assurée par un seul autre sémaphore `sem` initialisé à 0.

Dans le code généré par le compilateur, chaque fonction OPi du moniteur est encadrée par le code suivant :

```
P(mutex)                // debut SC -> on s'assure de l'exclusion mutuelle
   ...
   code pour OPi
   ...
V(mutex)               // Fin SC -> on  libère le moniteur
```

Pour chaque opération effectuée sur une variable condition, le compilateur doit générer du code particulier. L'opération `wait` génère le code suivant :

```
V(mutex)     // Libère le moniteur
P(sem)       // processus se bloque
P(mutex)     // processus re-demande l'entrée dans le moniteur
```

L'opération `signal` génère le code suivant :

```
 V(sem)
```

Implication au niveau de la programmation

Le mode de fonctionnement du `notify` (par rapport au `signal`) implique un changement majeur au niveau de la programmation basée sur ces variables condition.
En effet, dans le cas des variables condition de type `wait/signal` présenté à la section 4.3.1, l'opération `wait` peut être incluse dans un simple énoncé de sélection de la manière suivante :

```
if (cond) .... wait().
```

Cela est possible car l'opération `signal` transfère immédiatement le contrôle au processus débloqué. Cette propriété offre des avantages importants du point de vue de la performance (on ne teste qu'un seule fois la condition donc un seul changement de contexte).
En adoptant les variables condition de type `wait/notify`, l'opération `wait` doit maintenant **obligatoirement** apparaître dans un énoncé d'itération de la manière suivante :

```
while (cond) .... wait()
```

Cela devient nécessaire car, lors de l'exécution d'un `notify`, le contrôle n'est pas transféré immédiatement au processus débloqué. Ce dernier entre donc en compétition avec tous les processus désirant entrer en section critique (ligne 3 du `wait`) et devra obligatoirement re-tester la condition. Le programme 4.20 implante une nouvelle version du tampon à n éléments basée sur cette version des variables condition (`wait/notify`).
Ainsi, cette implantation risque d'entraîner de multiples changements de contexte. Malgré tout, elle s'est imposée car elle est plus simple à implanter et respecte mieux les priorités du système. En effet, si la file d'attente du sémaphore `mutex` respecte les priorités, le processus ayant la plus haute priorité sera toujours le premier à accéder au moniteur. Stallings [11] présente les avantages et inconvénients des deux approches.

4.3.5 Autres opérations sur les variables conditions

De nombreux langages ont introduit de nouvelles opérations sur les variables conditions :

- `Signal/wait` : leur comportement a déjà été présenté (Hoare).
- `Signal/continue` : leur comportement a déjà été présenté (Mesa).
- `Signal/exit` : leur comportement est le même que celui des variables de type `queue`.
- `Signal/Urgent Wait` : le processus débloqué par l'opération `signal` prend le contrôle immédiatement et le processus ayant émis le `signal` reprend ensuite le contrôle en priorité.
- `SignalAll` : tous les processus bloqués sont libérés.
- `Signal/Wait(délai)` : le processus ayant initié le `wait` est débloqué après le délai indiqué (en Java).

4.3.6 Évaluation

Les moniteurs sont parmi les outils les plus utilisés. Ils sont aussi parmi les plus critiqués.
Voici quelques thèmes de discussions à leur sujet :

- Les appels de moniteurs imbriqués ;
  Si un moniteur $M1$ appelle un moniteur $M2$, le processus doit-il relâcher son droit d'accès exclusif sur M1 (un fois dans M2) ? La solution ne fait pas l'unanimité. En effet, si l'exclusion mutuelle n'est pas relâchée, la performance risque d'en être affectée en plus d'introduire des

```
1  type tampon(T) = monitor
2  var  item :  array[0..n-1] of T;
3       tete, queue : 0.. n-1;
4       dimension : 0..n;
5       non-plein, non-vide : condition;
6
7  procedure deposer(p : T);
8    begin
9       while dimension = n then non-plein.wait;
10        item[queue] := p;
11        dimension := dimension + 1;
12        queue := (queue + 1) mod n;
13        non-vide.notify;
14   end;
15 procedure retirer(var p : T);
16   begin
17      while dimension = 0 then non-vide.wait;
18      p := item[tete];
19      dimension := dimension - 1;
20      tete := (tete + 1) mod n;
21      non-plein.notify;
22   end;
23 begin
24    dimension := 0; tete := 0; queue := 0;
25 end;
```

Programme 4.20 – Implantation d'un type tampon avec un moniteur (wait/notify)

possibilités d'interblocage. En revanche, le fait de relâcher l'exclusion mutuelle peut impliquer la perte de la ressource et de l'information (possiblement incomplète) qui y était emmagasinée.

- L'ordre de sortie des variables conditions ;
  Quel sera le prochain processus débloqué ? Une politique de type FIFO est très simple mais elle ne convient pas à toutes les situations. En particulier, lorsque notre choix est de recourir aux priorités. Hoare a donc introduit un paramètre p à l'opération `wait` qui devient désormais :

  `cond.wait(p)`

  où p est une expression entière évaluée lors de l'exécution du `wait`. La valeur de p, appelée numéro de priorité, est emmagasinée avec le nom du processus suspendu. Lors du signal, le premier processus libéré est alors celui qui possède la plus petite valeur de p.
  Par exemple, dans le programme 4.21, ce paramètre est employé pour implanter un gestionnaire de ressource selon la politique SJF (le plus court d'abord). Lors d'un appel à ce gestionnaire, un processus fournit un estimé du temps d'utilisation de la ressource. La valeur du paramètre p est calculée à partir de cet estimé. Ainsi, le processus ayant la plus petite valeur de p (le temps le plus court) obtiendra la ressource en premier.
- La modularité ;
  Les opérations fournies par un moniteur se font en exclusion mutuelle implicitement (aucun énoncé spécifique pour y parvenir). Toutefois, la synchronisation conditionnelle s'effectue à l'aide d'énoncés sur des variables conditions dispersés dans le code de la fonction.
  En conséquence une partie du code de synchronisation est «invisible» alors que l'autre est «visible». Certains sont favorables à ce que toutes les parties du code de synchronisation soient

```
type SJF = monitor
     var libre : boolean;
         tour  : condition;
procedure demander(time : integer);
   begin
      if not libre then tour.wait(time);
      libre := faux;
   end;
procedure libere();
   begin
      libre := vrai;
      tour.signal;
   end;
begin
  libre := vrai;
end;
```

Programme 4.21 – Gestionnaire de ressources utilisant la politique SJF

invisibles ou, tout le moins, qu'elles ne soient pas dispersées dans le code de la fonction.

- L'exclusion mutuelle «universelle» ;
  Dans certaines situations, il serait utile que certaines opérations du moniteur ne s'exécutent pas en exclusion mutuelle.

4.4 Les moniteurs étendus (Crowd Monitor) [4]

L'usage de moniteurs comporte la contrainte d'exclusion mutuelle pour toutes les fonctions. Rappelons cependant que l'exclusion mutuelle ne doit jamais être conservée sur une longue période sans quoi, elle nuit au bon fonctionnement du système (cela limite parfois inutilement la concurrence et nuit à la performance).

Dans un moniteur, l'exclusion mutuelle s'avère parfois longue. Par exemple, dans le problème des lecteurs/écrivains, les opérations de lectures et d'écritures ont parfois une durée significative. De plus, si l'on situe ces opérations à l'intérieur du moniteur, les lectures simultanées deviennent impossible. Pour les permettre, ces dernières doivent se situer impérativement à l'extérieur du moniteur. Dans ce cas, le moniteur ne protège plus entièrement les données partagées, car l'opération de lecture est susceptible d'être lancée directement sans passer par le moniteur.

La solution proposée est l'introduction des moniteurs étendus. Ce type de moniteur permet de définir

- des procédures gardées exécutées en exclusion mutuelle ;
- des procédures normales soumises à des appels provenant exclusivement des processus autorisés. Les autorisations sont distribuées dans les procédures gardées.

Le programme 4.22 reprend le problème des lecteurs/écrivains en utilisant les moniteurs étendus. Le contrôle de l'accès aux opérations `read` et `write` est fait grâce à l'énoncé `enter`. Ainsi, pour obtenir le droit d'appeler l'opération `read`, un processus doit faire partie du groupe des `lecteurs`. De même, pour appeler les opérations `write` et `read`, un écrivain doit faire partie du groupe `ecrivains`. Un validation dynamique à l'exécution s'assure du respect de ces règles.

```
1  crowd monitor readwrite;
2     export startread, endread, read, startwrite, endwrite, write;
3
4     var lecteurs :  crowd read;
5         ecrivains :  crowd read, write;
6
7     guard procedure startread;
8        begin;
9           ... (bloque l'appelant jusqu'à autorisation pour lire)
10          enter lecteurs;
11       end;
12    guard procedure endread;
13       begin
14          leave lecteurs;
15          ... (libération de la ressource si nécessaire)
16       end};
17
18    guard procedure startwrite;
19       begin
20          ... (bloque l'appelant jusqu'à autorisation pour écrire)
21          enter ecrivains;
22       end;
23    guard procedure endwrite;
24       begin
25          leave ecrivains;
26          ... (libération de la ressource)
27       end;
28
29    procedure read;
30       begin
31          ... (lecture de la ressource partagée);
32       end;
33    procedure write;
34       begin
35          ... (écriture de la ressource partagée) ;
36       end;
37    end readwrite;
```

Programme 4.22 – Solution au problème des lecteurs/écrivains basée sur les moniteurs étendus

4.5 Les expressions de chemins (Path Expressions) [1, 4]

Les expressions de chemins adoptent une approche différente de celles que nous avons présentées jusqu'à maintenant. Avec cette nouvelle construction, on spécifie tous les besoins en synchronisation à un seul endroit afin qu'ils soient autant que possible indépendants des opérations. On recherche donc une séparation complète entre l'implantation des opérations et les contraintes de synchronisation. Le compilateur sera en charge de générer automatiquement tout le code nécessaire à la synchronisation.

Les expressions de chemins sont basées sur l'utilisation d'une structure similaire à une classe ou à un moniteur. Les expressions de chemins, située dans l'entête de la classe, définissent toutes les contraintes sur l'ordre d'exécution des opérations (la synchronisation). Aucune contrainte de synchronisation n'est spécifiée à l'intérieur des procédures.

4.5.1 Syntaxe

Les expressions de chemins nécessitent une syntaxe précise faisant appel à un certain nombre d'opérateurs de chemins. Le format général d'une expression de chemins est :

`PATH` *liste* `END;`

La *liste* contient les noms des opérations déclarées dans la classe et ceux des opérateurs requis pour construire le chemin. On dispose de quatre opérateurs pour définir un chemin :

1. « `,` » : indique la concurrence, i.e. que les opérations ont la possibilité de s'exécuter en parallèle. Par exemple,

   «`PATH op1,op2 END`»

   indique qu'il est possible d'exécuter les opérations `op1` et `op2` simultanément.

2. « `;` » : indique la séquence à respecter dans l'exécution des opérations. Par exemple,

   «`PATH op1;op2 END`»

   indique que l'opération `op1` doit obligatoirement s'exécuter avant l'opération `op2`.

3. « `n:(liste)` » : indique que `n` exécutions concurrentes de la liste peuvent coexister (`n` exécutions parallèles).

4. « `[liste]` » : indique qu'un nombre illimité d'exécutions concurrentes de la liste peuvent coexister

Voici quelques exemples d'expressions de chemins implantant un tampon contenant N éléments :

- `PATH deposer, retirer` END ;

  Selon cette spécification, il n'existe aucune contrainte sur l'exécution de ces opérations, autant sur l'ordre d'exécution que sur le nombre d'activations concurrentes de chaque opération. Dû au fait qu'il n'y a aucune synchronisation, le risque de corruption du tampon est présent.

- `PATH [deposer],[retirer]` END ;

  Cette spécification est équivalente à la précédente. Le «[ ]» indique seulement de façon explicite qu'il peut y avoir une infinité d'activations de ces opérations en parallèle.

- `PATH [deposer,retirer]` END ;

  Cette spécification est équivalente à la précédente. Le «[ ]» indique seulement de façon explicite

qu'il peut y avoir une infinité d'activations de la liste en parallèle.

- `PATH deposer;retirer END;`
  Cette spécification indique que toutes les opérations «`retirer`» doivent être précédées d'une opération «`deposer`». Il peut cependant se produire de multiples activations de la liste «`deposer;retirer`» (donc de multiples exécutions parallèles de chaque opération) en autant que le nombre de `retirer` actif ou complété ne dépasse jamais le nombre de `deposer` complété.
  Selon cette spécification, une opération `retirer` ne pourra lire un élément qui n'aura pas été préalablement déposé.
- `PATH 1:(deposer;retirer) END;`
  Selon cette spécification, on implante un tampon contenant un seul élément. En effet, on indique qu'une seule exécution de la liste doit se faire à la fois et que celle-ci se fera dans l'ordre `deposer` puis `retirer`. Les exécutions des `deposer` et `retirer` alternent de façon stricte en exclusion mutuelle.
- Spécification correcte de la concurrence sur un tampon contenant N éléments (où $n > 1$) ?
  Pour obtenir une spécification correcte de ce tampon, il faut :
  - pouvoir produire au plus N éléments en avance sur les retraits (le nombre de `deposer` complété est au plus de N supérieur au nombre de `retirer` complété) ;
  - éviter de retirer des éléments qui n'ont pas été produits (chaque activation de `retirer` est précédée par un `deposer` complété) ;
  - que les opérations `deposer` s'exécutent en exclusion mutuelle ;
  - que les opérations `retirer` s'exécutent en exclusion mutuelle.

  L'expression de chemins suivante implante ces contraintes :

  PATH `n:((1:deposer);1:(retirer))` **END**

  L'implantation complète du tampon utilisé par notre petit système d'exploitation avec traitement par lots est fournie par le programme 4.23. Il est important de noter que les `deposer` et `retirer` peuvent s'exécuter en parallèle. Cela était impossible avec les moniteurs. De plus, on remarque une séparation complète de l'implantation des opérations et de la synchronisation.
- Le programme 4.24 implante un sémaphore avec les expressions de chemins. Cet exemple fournit un sémaphore général qui, comme on peut le constater, est «initialisé» à 0 par défaut.

  > Est-il possible de l'initialiser à une autre valeur que 0 ?

- Pour terminer, voici quelques solutions proposées au problème des lecteurs/écrivains qui font appel aux expressions de chemins. La difficulté ici est d'exprimer correctement les priorités.
  1. Cette première solution (présentée ci-dessous) semble donner la priorité aux lecteurs, toutefois ce n'est pas tout à fait le cas. En effet, si un écrivain prend le contrôle, tous les écrivains en attente dans la file (s'il n'y a pas de lecteurs) pourront passer avant les nouveaux lecteurs. Ce n'est donc pas une priorité absolue.

```
class tampon(T) =  module
PATH n:((1:deposer);1:(retirer)) END

var  item : array[0..n-1] of T;
     tete, queue : 0.. n-1;
procedure deposer(p : T);
   begin
       item[queue] := p;
       queue := (queue + 1) mod n;
   end;
procedure retirer(var p : T);
   begin
       p := item[tete];
       tete := (tete + 1) mod n;
   end;
begin
    tete := 0; queue := 0;
end;
```

Programme 4.23 – Implantation d'un tampon à N éléments avec les expressions de chemins.

```
PATH [V;P] END

procedure V()
  begin
  end;

procedure P()
  begin
  end;
```

Programme 4.24 – Implantation d'un sémaphore avec les expressions de chemins.

PATH 1 :([read],write) **END**

2. La seconde solution tente de donner une priorité aux écrivains. Elle ne leur donne cependant pas une priorité absolue car une fois qu'une lecture «passe», toutes les lectures «passent» automatiquement.

PATH 1 :([read],[WRITE]) **END**
PATH 1 :(write)) **END**

WRITE= **begin** write **end**;

3. La troisième solution donne une priorité absolue aux lecteurs. Pour y parvenir, l'introduction de fonctions intermédiaires a été nécessaire. Ces fonctions ne servent, en fait, qu'à assurer la synchronisation.

PATH 1 :(write-attempt) **END**
PATH 1 :([request-read],request-write) **END**
PATH 1 :([read],(open-write ;write)) **END**

write-attempt = **begin** request-write **end** ;
request-write = **begin** open-write **end** ;
request-read = **begin** read **end** ;

READ = **begin** request-read **end** ;
WRITE = **begin** write-attempt ; write **end** ;

4. La quatrième solution donne priorité aux écrivains. On vous laisse la déchiffrer.

PATH 1 :(read-attempt) **END**
PATH 1 :(request-read,[request-write]) **END**
PATH 1 :([open-read ;read],write) **END**

read-attempt = **begin** request-read **end** ;
request-read = **begin** open-read **end** ;
request-write = **begin** write **end** ;

READ = **begin** read-attempt, read **end** ;
WRITE = **begin** request-write **end** ;

5. La cinquième solution permet de répondre aux lecteurs et aux écrivains dans l'ordre d'arrivée. Les lectures qui sont consécutives dans le file pourront être servies simultanément.

PATH 1 :(request-read,request-write) **END**
PATH 1 :([open-read ;read], write) **END**

request-read = **begin** open-read **end** ;
request-write = **begin** write **end** ;

READ = **begin** request-read, read **end** ;
WRITE = **begin** request-write **end** ;

4.5.2 Implantation

Les expressions de chemins sont traduites directement par une suite d'opérations `P` et `V` sur des sémaphores. Le compilateur fournissant des expressions de chemins doit générer ces opérations en

prologue ou en épilogue des procédures définies dans la classe, et ce, afin d'assurer automatiquement la synchronisation.

La traduction se fait en plusieurs étapes :

1. À partir de $PATH < liste > END$, génération de trois listes L, M et R où :
   - L est la liste composée des éléments de gauche. Initialement, elle est vide, $L = \{\}$ car $PATH$ ne génère rien [2].
   - M est la liste composée de l'élément central. Initialement, elle contient la liste, $M = \{liste\}$.
   - R est la liste composée des éléments de droite. Initialement, elle est vide, $R = \{\}$ car END ne génère rien.

   Toutes les listes sont décomposées en «$L\ M\ R$» où le nouveau «L», est l'élément de gauche et le nouveau «R», l'élément de droite. L'élément «M» contient toujours le reste de la liste à décomposer. À la fin, lorsque la liste centrale ne contient plus d'opérateurs de chemins, «L» contiendra le code à ajouter en prologue à chaque fonction et «R» celui à ajouter en épilogue.

2. Examen de la liste contenue dans M.

   - Si $L < exp_1 >, < exp_2 > R \rightarrow$ division de la liste en deux :
     (a) $L < exp_1 > R$
     (b) $L < exp_2 > R$

     Dans les deux cas, L et R demeurent inchangés.

   - Si $L < exp_1 >; < exp_2 > R$ alors :
     - Création d'un sémaphore $s1 = 0$
     - Division de la liste en deux :
       (a) «$L < exp_1 > R'$» où $R' = \{V(s1)\}$
       (b) «$L' < exp_2 > R$» où $L' = \{P(s1)\}$

   - Si $L\ \ n : (< exp >)\ \ R$ alors :
     - Création d'un sémaphore $S2 = n$
     - Modification des composants L et R afin que la liste devienne :
       (a) «$L' < exp > R'$» où $L' = \{P(S2) \cup L\}$ et $R' = \{R \cup V(S2)\}$

2. Il existe des implantations d'expressions de chemins dans lesquelles $PATH < liste > END$ génère automatiquement un accès exclusif à la liste.

- Si L $[< exp >]$ R alors :
  - Création d'un sémaphore $S3 = 1$ et d'un entier $c = 0$;
  - Création des fonctions suivantes :

```
PP(c,s3,L)
{
  P(s3)
  c := c + 1;
  if c = 1 the L;
  V(s3)
}
```

```
VV(c,s3,RL)
{
  P(s3)
  c := c - 1;
  if c = 1 then R;
  V(s3)
}
```

  - Modification des composants L et R afin que la liste devienne :
    (a) $L' < exp > R'$
    où $L' = \{PP(c, s3, L)\}$ et $R' = \{VV(c, s, R)\}$

- Si «$L <$ *opération* $> R$» génération de la fonction et du code :

```
begin
  L;
  <operation>;
  R;
end;
```

4.5.3 Exemples de traductions

Exemple 1 : PATH deposer, retirer END

Soit l'expression de chemins suivante : **PATH deposer, retirer END**

1. Au départ, on a :
   $$L = \{\}, R = \{\} \text{ et } M = deposer, retirer$$
2. On décompose $M = deposer, retirer$ et on obtient :
   - L *deposer* R
     où et $L = \{\}$ et $R = \{\}$
   - L *retirer* R
     où et $L = \{\}$ et $R = \{\}$
3. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  deposer();
end;
```

```
begin
  retirer();
end;
```

Exemple 2 : PATH deposer ; retirer END

Soit l'expression de chemins suivante : **PATH deposer ; retirer END**

1. Au départ, on a :
$$L = \{\},\ R = \{\} \text{ et } M = deposer; retirer$$
2. On décompose $M = deposer; retirer$ et on obtient :
   - L' *deposer* R'
     où et $L' = L = \{\}$ et $R' = R \cup V(s1) = \{V(s1)\}$
   - L' *retirer* R'
     où et $L' = L \cup P(s1) = \{P(s1)\}$ et $R' = R = \{\}$
3. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  deposer();
  V(s1);
end;
```

```
begin
  P(s1);
  retirer();
end;
```

Exemple 3 : PATH 1 :(deposer ;retirer) END

Soit l'expression de chemins suivantes : **PATH 1 :(deposer ;retirer) END**

1. Au départ, on a :
$$L = \{\},\ R = \{\} \text{ et } M = 1 : (deposer; retirer)$$
2. On décompose $M = 1 : (...)$ et on obtient :
   - L' *deposer*; *retirer* R'
     où $L' = L \cup P(s1) = \{P(s1)\}$ et $R' = R \cup V(s1) = \{V(s1)\}$
3. On décompose $M = deposer; retirer$ et on obtient :
   - L' *deposer* R'
     où et $L' = L = \{P(s1)\}$ et $R' = \{V(s2)\}$
   - L' *retirer* R'
     où et $L' = \{P(s2)\}$ et $R' = \{V(s2)\}$
4. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  P(s1);
  deposer();
  V(s2);
end;
```

```
begin
  P(s2);
  retirer();
  V(s1);
end;
```

Exemple 4 : PATH [deposer ; retirer] END

Soit l'expression de chemins suivante : **PATH [deposer ; retirer] END**

1. Au départ, on a :
$$L = \{\},\ R = \{\} \text{ et } M = [deposer, retirer]$$
2. On décompose $M = 1 : (...)$ et on obtient :
   - L' $deposer, retirer$ R'
     où $L' = \{PP(c, s, L)\}$ et $R' = \{VV(c, s, R)\}$
3. On décompose $M = deposer, retirer$ et on obtient :
   - L' $deposer$ R'
     où et $L' = \{PP(c, s, L)\}$ et $R' = \{VV(c, s, R)\}$
   - L' $retirer$ R'
     où et $L' = \{PP(c, s, L)\}$ et $R' = \{VV(c, s, R)\}$

   Comme L et R sont vides, on peut éliminer les fonctions PP et VV.
4. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  deposer();
end;
```

```
begin
  retirer();
end;
```

Exemple 5 : PATH 1 :([lecture] ;écriture) END

Soit l'expression de chemins suivante : **PATH 1 :([lecture] ;écriture) END**

1. Au départ, on a :
$$L = \{\},\ R = \{\} \text{ et } M = 1 : ([lecture], ecriture)$$
2. On décompose $M = 1 : (...)$ et on obtient :
   - L' $[deposer], retirer$ R'
     où $L' = L \cup P(s1) = \{P(s1)\}$ et $R' = R \cup V(s1) = \{V(s1)\}$
3. On décompose $M = [deposer]; retirer$ et on obtient :
   - L' $[deposer]$ R'
     où et $L' = L = \{P(s1)\}$ et $R' = R = \{V(s1)\}$
   - L' $retirer$ R'
     où et $L' = \{P(s1)\}$ et $R' = \{V(s1)\}$
4. On décompose $M = [deposer]$ et on obtient :
   - L' $deposer$ R'
     où et $L' = \{PP(c, s, L)\}$ et $R' = \{VV(c, s, R)\}$
5. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  PP(c,s,P(s1));
  deposer();
  VV(c,s,V(s1));
end;
```

```
begin
  P(s1);
  retirer();
  V(s1);
end;
```

Exemple 6 : PATH n :(1 :(lecture) ;1 :(écriture)) END

Soit l'expression de chemins suivante : **PATH n :(1 :(lecture) ;1 :(écriture)) END**

1. Au départ, on a :
$$L = \{\},\ R = \{\} \text{ et } M = n : (1 : (deposer); 1 : (retirer))$$
2. On décompose $M = n : (...)$ et on obtient :
   - L' *deposer*; *retirer* R'
     où $L' = L \cup P(s1) = \{P(s1)\}$ et $R' = R \cup V(s1) = \{V(s1)\}$
3. On décompose $M = 1 : (deposer); 1 : (retirer)$ et on obtient :
   - L' *deposer* R'
     où et $L' = L = \{P(s1)\}$ et $R' = \{V(s2)\}$
   - L' *retirer* R'
     où et $L' = \{P(s2)\}$ et $R' = \{V(s2)\}$
4. On décompose $M = 1 : (deposer)$ et on obtient :
   - L' *deposer* R'
     où $L' = L \cup P(s1) = \{P(s2); P(s1)\}$ et $R' = R \cup V(s1) = \{V(s2); V(s1)\}$
5. On décompose $M = 1 : (retirer)$ et on obtient :
   - L' *deposer*; *retirer* R'
     où $L' = L \cup P(s1) = \{P(s2); P(s1)\}$ et $R' = R \cup V(s1) = \{V(s2); V(s1)\}$
6. Comme il n'y a plus d'opérateurs de chemins, on génère le code suivant :

```
begin
  P(s2);
  P(s1);
  deposer();
  V(s2);
  V(s1);
end;
```

```
begin
  P(s1);
  P(s2);
  retirer();
  V(s1);
  V(s2);
end;
```

4.6 Les expressions invariantes [8, 4]

L'approche de synchronisation basée sur les expressions invariantes est orientée sur le même principe que sur celui des expressions de chemins, soit regrouper toutes les contraintes de synchronisation au même endroit.

Chacune des activités qui tente d'exécuter une procédure est associée à une liste. Des expressions invariantes sont associées à chacune de ces listes et servent à limiter l'accès à la procédure. En particulier, ce sont ces expressions qui déterminent si un processus en attente peut ou non exécuter la procédure. Notons que, s'il n'y a aucune expression invariante associée à une procédure, c'est qu'il n'y a aucune restriction sur son exécution.

Les expressions invariantes possèdent une syntaxe relativement limitée. Elles requièrent des compteurs prédéfinis qui enregistrent les événements pertinents concernant les différentes procédures. Soit une procédure «`proc`», on lui associe les cinq compteurs suivants :

- requestCount(proc) : contient le nombre d'appels totaux à la procédure ;
- startCount(proc) : contient le nombre de fois que la procédure a démarré son exécution ;
- finishCount(proc) : contient le nombre de fois que la procédure a terminé son exécution ;
- currentCount(proc) = startCount(proc) - finishCount(proc) : contient le nombre d'exécutions en cours (concurrentes) ;
- waitCount(proc) = requestCount(proc) - startCount(proc) : contient le nombre d'appels en attente.

Les expressions invariantes sont des expressions arithmétiques contenant des sommes ou des soustractions sur les compteurs. Elles ont la forme suivante :

expression comparaison constante

Les comparaisons utilisent les opérateurs $<, >, \leq, \geq, =, \neq$. Voici donc un exemple d'une expression invariante :

waitCount(proc1) - requestCount(proc2) > 4

Ces expressions sont associées aux procédures et indiquent les contraintes à satisfaire avant d'autoriser leur exécution.

4.6.1 Exemples

Le programme 4.25 présente une solution aux problèmes du tampon à N éléments. Il est intéressant de remarquer qu'ici, les procédures ne contiennent aucun code de synchronisation. De plus, les expressions invariantes ont l'avantage d'être plus lisibles que les expressions de chemins.

Le programme 4.26 introduit deux solutions au problème des lecteurs/écrivains. La première donne une priorité aux lecteurs et la seconde aux écrivains. Notons aussi qu'il est pratiquement impossible d'implanter une solution de type premier arrivé, premier servi, avec les expressions invariantes.

```
type tampon(T) = module
   export : deposer, retirer;

   var  item : array[0..n-1] of T;
        tete, queue : 0.. n-1;

invariant deposer
     startcount(deposer) - finishCount(retirer) $<$ n;
     currentCount(deposer) = 0;
invariant retirer
     startcount(retirer) - finishCount(deposer) $<$ 0;
     currentCount(retirer) = 0;

procedure deposer(p : T);
    begin
       item[queue] := p;
       queue := (queue + 1) mod n;
    end;
procedure retirer(var p : T);
    begin;
       p := item[tete];
       tete := (tete + 1) mod n;
    end;
begin
     tete := 0; queue := 0;
end;
```

Programme 4.25 – Tampon à N éléments avec des expressions invariantes

```
1.   invariant read
          currentCount(write) = 0;
     invariant write
          currentCount(read) + currentCount(write) = 0;


2.   invariant read
          waitCount(write) + currentCount(write) = 0;
     invariant write
          currentCount(read) + currentCount(write) = 0;
```

Programme 4.26 – Solution au problème des lecteurs/écrivains

4.7 Les compteurs d'événements

Les compteurs d'événements, introduits par Reed et Kanadia[7], consistent en une structure de contrôle abstraite. Ils ont été présentés comme une alternative aux sémaphores principalement pour les systèmes distribués. Ils permettent aux processus de contrôler l'ordonnancement des événements afin de protéger les manipulations des variables partagées, plutôt que de recourir à l'exclusion mutuelle.

Un compteur d'événements est un objet contenant l'information sur le nombre d'événements pertinents qui se sont produits dans le système. C'est une variable entière dont la valeur ne peut décroître. Soit un compteur d'événements E, trois opérations sont définies sur celui-ci :

- `avance(E)` : Cette opération permet de signaler l'occurrence d'un événement associé au compteur. Il incrémente le compteur. Il réveille aussi tous les processus en attente de cette valeur nouvellement atteinte par E.
- `lire(E)` : Cette opération permet d'obtenir la valeur courante du compteur.
- `attendre(E,v)` : Cette opération permet de bloquer l'exécution d'un processus jusqu'à ce que la valeur du compteur soit égale à v.

Le programme 4.27 implante une solution basée sur les compteurs d'événements qui résout le problème des producteur/consommateur se partageant un tampon de N éléments. Le tampon est circulaire et les éléments sont indicés de 0 à $N-1$. Les compteurs d'événements IN et OUT, initialisés à 0, sont utilisés pour synchroniser le producteur et le consommateur. Notons que cette solution n'est valide que pour les situations n'ayant qu'un seul producteur et un seul consommateur.

```
1  eventcount IN:=0, OUT :=0
2  procedure producteur()
3      begin
4          var i: int
5          for i:=0 to infinie do
6              begin
7                  produire (p)
8                  attendre(OUT, (i-N)+1)
9                  tampon[i mod N] := p;
10                 avance(IN)
11             end;
12     end;
13 procedure consommateur()
14     begin
15         var i: int
16         for i:=0 to infinie do
17             begin
18                 attendre(IN, i+1)
19                 consomme(tampon[i mod N])
20                 avance(OUT)
21             end;
22     end;
```

Programme 4.27 – Solution au problème des producteur/consommateur utilisant les compteurs d'événements

4.8 Les séquenceurs

Un séquenceur, aussi amené par Reed et Kanadia[7], est un objet constitué d'un entier sur lequel on définit une seule opération, soit `ticket(S)`, où S est une variable de type séquenceur. Cette opération produit de façon atomique une valeur unique v et ensuite incrémente la valeur du séquenceur (voir programme 4.28). Contrairement aux compteurs d'événements, un séquenceur permet d'obtenir une valeur unique. Il sert donc à implanter l'exclusion mutuelle comme au programme 4.29.

Le programme 4.30 décrit une solution au problème des producteurs/consommateurs dans lequel plusieurs producteurs et plusieurs consommateurs sont présents.

```
1 int ticket(S)
2 { // Exécute de façon atomique les énoncés suivants
3   v = seq.val;
4   seq.val++;
5   return v;
6 }
```

Programme 4.28 – Opération Ticket(S)

```
1 // Soit E un compteur d'événements et S un séquenceur
2 while (true) {
3     v = ticket(S);
4     attendre(E, v);
5     ... Section critique
6     avance(E);
7 }
```

Programme 4.29 – Exlusion mutuelle avec un compteur d'événements et un séquenceur.

Le programme 4.31 propose une solution au problème des lecteurs/écrivains avec priorité aux écrivains et ayant un seul lecteur (appelé l'observateur). Ce dernier tente de lire une donnée. Si la donnée est modifiée entre deux lectures, alors il considère qu'il accédait à la donnée en même temps qu'un écrivain. Il recommence donc jusqu'à ce qu'il n'y ait aucune écriture pendant sa lecture.

4.9 Les «sérialiseurs»

Un sérialiseur (serializer) est une structure similaire au moniteur. C'est un type abstrait de donnée qui inclut un ensemble de procédures et qui encapsule les ressources partagées. Comme un moniteur, seulement un processus peut avoir accès à un sérialiseur à un moment donné.

Cette structure est basée sur le modèle des acteurs. Un sérialiseur est analogue à un poste d'accueil d'un hôpital au sens où seulement une personne est en mesure de vérifier les entrées et les sorties à la fois. Le poste d'accueil planifie les entrées et les sorties des patients de l'hôpital. Plusieurs places sont disponibles dans la salle d'attente pour les patients qui attendent leur tour afin qu'ils ne monopolisent pas la réception.

```
int buffer[N];
Prod_EV = new EventCount();    // Compte le nombre d'éléments produits
Cons_EV = new EventCount();    // Compte le nombre d'éléments consommés
Prod_SQ = new Sequencer();     // pour exclusion mutuelle
Cons_SQ = new Sequencer();     // pour exclusion mutuelle
process producteur(i:=1 to n)
  while(true)
  {
     item = produire();
     tour = ticket(Prod_SQ);
     attendre(Prod_EV, tour);
     attendre(Cons_EV, (tour-N)+1);
     buffer[tour % N] = item;
     avance(Prod_EV);
  }
}
process consommateur(j:=1 to n))
  while(true)
  {
     tour = ticket(CSQ);
     attendre(Cons_EV, tour);
     attendre(Prod_EV, tour+1);
     item = buffer[tour % N];
     avance(Cons_EV);
     consomme(item);
  }
}
```

Programme 4.30 – Solution au problème des producteurs/consommateurs

```


ecrivain_EV = new EventCount();  // compte le nbr de mises à jours (writes)
ecrivain_SQ = new Sequencer();   // pour exlusion mutuelle
lecteur_EV = new EventCount();   // version des données???

process ecrivain(i:=1 to M)
{
    avance(lecteur_EV);
    tour = ticket(ecrivain_SQ);
    attendre(ecrivain_EV, tour);
    ... faire la mise à jour
    avance(ecrivain_EV);
}
process lecteur(i:=1 to N)
{
   do
   {
       v1 = lecture(lecteur_EV);
       attendre(ecrivain_EV, v1);
       .. lire la donnée
       v2 = lecture(lecteur_EV);
   } while(v1 != v2);
```

Programme 4.31 – Solution au problème des lecteurs/écrivains

4.10 Les verrous et variations des sémaphores

4.10.1 Les verrous

En 1968, Dijkstra a créé les verrous comme mécanisme de synchronisation à utiliser dans le noyau du système. Il est possible d'exécuter seulement deux opérations sur un verrou V :

- `Lock(V)` qui prend le verrou ;
- `Unlock(V)` qui libère le verrou.

4.10.2 Variation des sémaphores

Sémaphore générique (chunk)

Les opérations sur un sémaphore générique permettent des modifications plus importantes sur la valeur du sémaphore. Ainsi, les opérations P et V sur un sémaphore générique S deviennent :

- `P(S,t)` dont l'effet est le suivant :

$$S \geq t \ ? \begin{cases} \text{Vrai} & \rightarrow \quad S := S - t, \\ \text{Faux} & \rightarrow \quad \text{bloque le processus} \end{cases}$$

- V(S,t) $\rightarrow S := S + t$

Sémaphore et disjonction

Ce type de structure combine plusieurs sémaphores en une seule opération. Soit un sémaphore S, les opérations deviennent :

- $\mathrm{P}_{or}(S_1, S_2, ...S_n)$ dont l'effet est le suivant :

$$S_1 > 0 \vee ... \vee S_n > 0 \ ? \begin{cases} \text{Vrai} & \rightarrow \quad S_j := S_j - 1, (\text{j est le plus petit index tel que } S_j > 0) \\ \text{Faux} & \rightarrow \quad \text{bloque le processus} \end{cases}$$

- $\mathrm{V}(S_i) \rightarrow S_i := S_i + 1$

Sémaphore et conjonction

Ce type de structure, comme la précédente, combine plusieurs sémaphores en une seule opération. Soit un sémaphore S, les opérations deviennent :

- $\mathrm{P}_{et}(S_1, S_2, ...S_n)$

$$S_1 > 0 \wedge ... \wedge S_n > 0 \ ? \begin{cases} \text{Vrai} & \rightarrow \quad S_1 := S_1 - 1; ...; S_n := S_n - 1 \\ \text{Faux} & \rightarrow \quad \text{bloque le processus} \end{cases}$$

- $\mathrm{V}(S_1, S_2, ...S_n) \rightarrow S_1 := S_1 + 1; ...S_n := S_n + 1$

Sémaphore générique et conjonction

- $\mathrm{P}_{et}(S_1, t_1; S_2, t_2; ...; S_n, t_n)$

$$S_1 \geq t_1 \wedge ... \wedge S_n \geq t_n \ ? \begin{cases} \text{Vrai} & \rightarrow \quad S_1 := S_1 - t_1; ...; S_n := S_n - t_n \\ \text{Faux} & \rightarrow \quad \text{bloque le processus} \end{cases}$$

- $\mathrm{V}(S_1, t_1; S_2, t_2; ...; S_n, t_n) \rightarrow S_1 := S_1 + t_1; ...S_n := S_n + t_n$

4.11 Autres mécanismes de synchronisation

Il existe plusieurs autres mécanismes moins populaires et moins connus :

- Predicate path expressions (à venir)
- Synchronization counters (à venir)
- Iteration expressions (Java) (à venir)
- ...

4.12 Évaluation

L'évaluation des outils de synchronisation se base sur les trois caractéristiques suivantes :

- la modularité

  La modularité d'un outil de synchronisation est rendue possible grâce à l'abstraction de données et à la séparation du contrôle de la concurrence de l'accès à la ressource.

  Toutes les techniques décrites offrent un bon degré de modularité. Notons toutefois que, dû aux au fait que les moniteurs ne forcent pas la séparation de la synchronisation de l'implantation des opérations, ceux-ci sont considérés moins modulaires que les expressions de chemins ou les expressions invariantes.

- la puissance d'expression

  La puissance d'expression est l'habileté à formuler l'exclusion mutuelle, la synchronisation conditionnelle et les besoins de priorité.

  Chacune des méthodes que nous avons présentées est capable de formuler l'exclusion mutuelle et la synchronisation conditionnelle. Aucune n'est apte à formuler correctement les besoins de priorité. En effet, cela leur est difficile ou impossible car les listes ne peuvent être ni inspectées, ni ré-ordonnées.

  Les expressions de chemins ont cependant moins de puissance d'expression que les autres. Il est en effet très complexe de déterminer la priorité sur le type de demande. Ainsi, il est difficile de résoudre le problème des lecteurs/écrivains en donnant la priorité aux écrivains ou aux lecteurs. Nous avons considéré plusieurs solutions qui permettent d'exprimer des priorités,

mais celles-ci sont complexes et exigent la création de fonctions «artificielles», i.e. des artifices pour déjouer le manque de puissance. Cela est dû au fait que les expressions de chemins sont limitées dans les types d'information qu'elles peuvent utiliser. Les distinctions ne se font que sur les types de demande. Lorsque les expressions de chemins ne sont pas en mesure d'exprimer une certaine contrainte, la possibilité d'implanter une solution en ajoutant des nouvelles procédures subsiste. Cependant, ces ajouts entraînent beaucoup trop d'interactions entre les procédures (appels) et va à l'encontre d'une implantation modulaire.

Ainsi, ce mécanisme, par manque de puissance d'expression, supporte mal la modularité.

Les expressions invariantes sont plus puissantes car elles ont accès à plus d'informations telles que les noms des procédures, leur historique d'appels et les appels en attente.

- la facilité d'utilisation

  La facilité d'utilisation est un critère subjectif. Les RCCs sont probablement plus conviviales que les moniteurs, mais elles sont moins fréquemment employées. Les expressions invariantes, elles, sont plus simples à utiliser que les expressions de chemins, ces dernières représentant probablement l'outil de synchronisation ayant la syntaxe la plus complexe (mais pas le plus puissant).

Le choix d'un mécanisme de synchronisation dépend de son efficacité, de sa facilité d'utilisation, mais surtout de sa disponibilité. Comme la plupart des langages fournissent des sémaphores et des «pseudo-moniteurs», ce sont les outils les plus populaires actuellement.

En ce qui concerne le noyau du système, l'approche par verrous de base est certainement la plus populaire.

Bibliographie

[1] R. H. Campbell et A. N. Habermann : The specification of process synchronization by path expressions. *In* E. Gelenbe et C. Kaiser, éditeurs : *Operating Systems - Proceedings of an International Symposium*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 89–102, Germany, 1974. Springer.

[2] Developpez : Optimisation des compilateurs. `https://algo.developpez.com/tutoriels/techniques/optimisation-des-compilateurs/`, 2014.

[3] Auteur externe : Les optimisations des compilateurs. `https://zestedesavoir.com/tutoriels/427/les-optimisations-des-compilateurs/ml`, 2018.

[4] Raphael A. Finkel : *An Operating Systems Vade Mecum : 2nd Edition*. Prentice-Hall, Inc., USA, 1988.

[5] Guy Grave : Les optimisation des compilateurs. `https://guy-grave.developpez.com/articles/optimisation-compilateurs/`, 2016.

[6] James L. Peterson et Abraham. Silberschatz : *Operating system concepts / James L. Peterson, Abraham Silberschatz*. Addison-Wesley Pub. Co Reading, Mass, 1983.

[7] David P. Reed et Rajendra K. Kanodia : Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, février 1979.

[8] Pierre Robert et Jean-Pierre Verjus : Toward autonomous descriptions of synchronization modules. *In Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, pages 981–986, 1977.

[9] Abraham Silberschatz, Greg Gagne et Peter Baer Galvin : *Operating System Concepts*. Wiley, 6 édition, 2002.

[10] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition*. Wiley, 2018.

[11] William Stallings : *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011.

[12] Wikibooks : Optimisation des compilateurs. `https://fr.wikibooks.org/wiki/Optimisation_des_compilateurs`, 2018.

[13] Wikipedia : Charles antony richard hoare. https://fr.wikipedia.org/wiki/Charles_Antony_Richard_Hoare, 2021.

[14] Wikipedia : Optimizing compiler. https://en.wikipedia.org/wiki/Optimizing_compiler, 2021.

[15] Wikipedia : Per brinch hansen. https://en.wikipedia.org/wiki/Per_Brinch_Hansen, 2021.