Précis de système d'exploitation
Synchronisation des processus

2ième Année Informatique et Mathématiques Appliquées
Gérard Padiou

4 septembre 2007

Table des matières

1 Introduction

Lorsque des activités parallèles s'exécutent, en particulier sous la forme de processus, il est nécessaire de contrôler l'utilisation des ressources mises en jeu. Toute activité consomme des ressources physiques (mémoire, processeur) ou logiques (programme,fichier), ressources qui sont d'une part limitées (espace mémoire par exemple) et d'autre part à partager entre toutes les activités. Ce partage peut prendre deux formes :

- plusieurs activités ont besoin de la même ressource mais celle-ci ne peut être utilisée correctement par plusieurs activités simultanément. Ce genre de ressource est dite **critique**. Par exemple, une imprimante ne peut être utilisée par plusieurs activités sous peine de mélanger les lignes résultats. Les activités entrent donc en conflit bien qu'à priori elles s'ignorent. Ce problème est un problème de compétition ou concurrence entre activités pour accéder aux ressources ;
- plusieurs activités partagent une ressource pour communiquer. Par exemple, une activité produit des messages qui seront consommés par une autre. Un tampon partagé servira de support pour mémoriser temporairement un message produit mais non encore consommé (lu). Ce problème est alors un problème de coopération entre des activités qui se coordonnent pour exécuter un traitement global en parallèle.

Dans ce qui suit, nous présentons un résumé des techniques et mécanismes proposés pour résoudre les problèmes de synchronisation. Pour une étude plus complète, on peut se reporter à [Pad90].

1.1 Hypothèses de base

Lorsque des activités interagissent entre elles, deux modèles d'interactions existent :

- le modèle **centralisé** dans lequel on suppose que les activités peuvent échanger des informations via une mémoire partagée. Autrement dit, les programmes exécutés partagent des variables globales ;
- le modèle **réparti** dans lequel on suppose que les activités peuvent échanger des informations via des communications par messages. Autrement dit, les programmes exécutés comportent des envois et/ou réception de messages supposés acheminés via un réseau.

Dans ce qui suit, nous nous intéressons au cas centralisé. Les activités partagent (peuvent partager) des variables globales.

Une deuxième hypothèse de base porte sur le comportement des activités. Là encore, il existe deux modèles :

- le modèle **synchrone** dans lequel toutes les activités parallèles exécutent leurs instructions élémentaires simultanément. Autrement dit, une horloge globale rythme toutes les actions des activités parallèles, actions qui ont de ce fait toutes la même durée. Certaines modèles synchrones tels qu'Esterel ou Lustre considèrent même que ces actions sont instantanées ;
- le modèle **asynchrone** dans lequel les activités progressent chacune à leur rythme. Ce modèle possède une complexité plus grande dans la mesure où il induit un non-déterminisme dans l'entrelacement des actions de chaque activité. Par suite, le nombre d'états possibles du système au cours de son exécution augmente exponentiellement et une exécution spécifique est difficilement **reproductible** (On a peu de chance de ré-exécuter les actions dans le même ordre compte tenu du non-déterminisme du modèle).

Dans ce qui suit, nous nous plaçons dans le cadre des systèmes asynchrones. Chaque activité exécute ses instructions élémentaires à son rythme. Une seul contrainte : toute activité (non terminée) doit exécuter une instruction infiniment souvent. Autrement dit, aucune activité non terminée ne reste indéfiniment passive.

En conclusion, on étudie donc la synchronisation d'activités parallèles dans un modèle centralisé asynchrone. Ce modèle correspond, de façon pratique, aux applications parallèles s'exécutant sur une architecture à mémoire commune mono ou multi-processeur(s), c'est-à-dire au cas le plus courant d'un système informatique.

1.2 Problème sémantique

Lorsque des activités parallèles s'exécutent, comment peut-on décider de la validité de leur exécution ? Deux cas sont à envisager : soit il y a finalement une terminaison globale de toutes les activités, soit les activités parallèles évoluent de façon réactive vis-à-vis d'un environnement sans atteindre un état stable définitif.

Pour exprimer la sémantique d'un programme séquentiel, nous utilisons la logique de Hoare. Rappelons que dans cette logique, la sémantique d'une instruction I s'exprime par une assertion ayant la forme d'un triplet :

$$\{P\}\ I\ \{Q\}$$

dans lequel le prédicat P représente une précondition et le prédicat Q une postcondition. Une telle assertion est vraie si et seulement si, l'exécution de l'instruction I à partir d'un état courant d'exécution vérifiant P conduit à un état vérifiant Q **si l'instruction se termine** (correction partielle).

Exemple Si x et y sont deux variables, la sémantique d'une instruction d'échange est décrite par le triplet :

$$\{x = a \wedge y = b\}\ swap(x, y)\ \{x = b \wedge y = a\}$$

Terminaison globale

Si les activités se terminent globalement, la validité du traitement exécuté peut reposer sur une comparaison par rapport au résultat que l'on aurait obtenu en exécutant séquentiellement les mêmes activités.

Il nous faut donc déterminer la postcondition correcte Q telle que

$$\{P\ \}\ \langle A_1\ ||\ A_2\rangle\ \{Q?\}$$

La solution consiste alors à accepter comme résultat correct de l'exécution parallèle tout résultat qui aurait été obtenu par une exécution séquentielle, c'est-à-dire ici, soit $A_1; A_2$ soit $A_2; A_1$. Autrement dit, la postcondition Q sera définie par $Q = Q_{12} \vee Q_{21}$ où Q_{12} et Q_{21} ont pour valeur :

$$\{P\}\ \langle A_1\ ;\ A_2\ \rangle\{Q_{12}\}, \{P\}\ \langle A_1\ ;\ A_2\rangle\ \{Q_{21}\}$$

Pas de terminaison

Dans ce cas, il faudra recourir à un invariant global permettant de capter la correction partielle du programme parallèle. À titre d'exemple, un schéma de communication de type producteur/consommateur entre deux processus pourra être spécifié par un invariant précisant que la séquence d'octets reçue par le consommateur est toujours un préfixe de la séquence émise par le producteur.

2 Compétition et concurrence

À travers deux exemples simples, nous mettons en évidence les problèmes de base pouvant survenir lorsque des processus partagent des variables (communes).

2.1 Exemple de concurrence (compétition)

Considérons un système de réservation de places dans une salle de spectacle, un avion, un train, ...etc Il faut gérer un compteur des places disponibles. Les transactions issues des différents points de réservation peuvent conduire à la modification de cette variable via la procédure de test et réservation :

```
bool reserver ( int p ) {
      if ( p > nDispo) return false;  /* test */
      nDispo = nDispo - p;            /* décrémentation */
      return true;
}
```

La sémantique d'une telle procédure en terme de triplets de Hoare est la suivante :

$$\{nDispo = k\}\ reserver(x)\ \{(k \geq x \Rightarrow nDispo = k - x) \wedge (k < x \Rightarrow nDispo = k)\}$$

Supposons que l'on exécute les deux réservations suivantes en parallèle :

$$\{nDispo = 12\}\ \langle reserver(10)||reserver(5)\rangle\ \{Q?\}$$

Le résultat final acceptable est celui obtenu par l'exécution de $\langle reserver(10); reserver(5)\rangle$ ou par l'exécution de $\langle reserver(5); reserver(10)\rangle$. Les deux exécutions conduisent à n'accepter qu'une seule des deux réservations (la première demandée) et les résultats acceptables sont donc :

$$\{nDispo = 12\}\ \langle reserver(10)||reserver(5)\rangle\ \{(nDispo = 7) \vee (nDispo = 2)\}$$

Cependant, l'exécution parallèle des deux réservations peut donner d'autres résultats. En effet, il suffit, par exemple, que les instructions de test des deux opérations de réservation se succèdent. Les deux tests trouvent la condition $(p > nDispo)$ fausse et par conséquent décrémentent le nombre de places disponibles. Finalement, la postcondition devient $\{nDispo = -3\}$ (il y a « sur-booking » !).

En fait, les instructions de test et décrémentation ne doivent pas s'entrelacer n'importe comment. À partir du moment où la variable partagée *nDispo* a été lue pour être comparée à la demande p, elle ne peut être lue par une autre opération *reserver* exécutée en parallèle tant que la première opération de réservation n'a pas décrémenté le nombre de places disponibles. La « lecture-modification » faite dans le cadre d'une réservation ne doit pas interférer avec tout autre opération. On dit que les opérations s'excluent mutuellement ou bien encore qu'elles doivent être exécutées en exclusion mutuelle. La séquence de code de la procédure *reserver* peut aussi être vue comme une ressource critique (appelée section critique) qui ne peut être utilisée par plus d'une activité à la fois. Autrement dit, toute transaction de réservation devra obtenir le droit d'exécution exclusif de la procédure. Attention à ne pas faire de confusion : cette propriété de criticité ne signifie pas que :

- pendant l'exécution de cette procédure critique, aucune autre activité parallèle ne peut s'exécuter ;
- que l'exécution de la procédure se déroule forcément en mode ininterruptible.

2.2 Exemple de coopération

Considérons une application de comptage de véhicules sur une route. Cette application se compose de deux activités cycliques :

- d'une part, un captage des événements « passage » de véhicule ;
- d'autre part, un affichage périodique (toutes les 10mns, toutes les 1/2 heures,...) du nombre de véhicules passés dans la période fixée.

Ce système parallèle se décrit sous la forme de deux processus partageant une variable globale compteur :

```
int cpt = 0 ; // variable partagée
process Capteur() {
  while (true) {
     attendre(passage);
     cpt++ ;
  }
}
```

```
process Consignateur() {
   while (true) {
       attendre(delai) ;
       printf("nbre passés = %i",cpt) ;
       cpt = 0 ;
   }
}
```

Nous sommes ici dans le cas d'un système réactif dont l'activité peut se poursuivre indéfiniment. Il faut donc trouver un invariant global permettant de contrôler la correction de ce système parallèle. Pour ce faire, on introduit une variable auxiliaire Tot qui permettra de vérifier si la somme des valeurs affichées est bien correcte. Le processus Capteur devient :

```
process Capteur() {
    int Tot = 0 ;
   while (true) {
      « attendre un passage »
      cpt++ ;   Tot++ ;
   }
}
```

Si l'on dénote par v_i les valeurs affichées jusqu'à l'état courant, l'invariant spécifiant un affichage correct du nombre de véhicules passés est le suivant :

$$\textbf{invariant } Tot = \sum_i v_i + cpt$$

Cet invariant doit être observable et toujours vrai lorsque les deux processus `Capteur` et `Consignateur` sont en attente. Or, tel n'est pas le cas. En effet, il suffit qu'une ou plusieurs incrémentations du compteur partagé aient lieu entre la lecture du compteur et sa remise à zéro par le consignateur pour que des passages se trouvent, de ce fait, non comptabilisés et donc mettent en défaut l'invariant puisqu'alors $Tot > (\sum_i v_i + cpt)$. En réalité, l'incrémentation du compteur d'une part et sa lecture et remise-à-zéro d'autre part sont des sections de code qui s'excluent mutuellement. On constate donc que le problème à résoudre est de même nature que pour l'exemple de compétition précédent : il faut garantir l'exclusion mutuelle entre différentes sections critiques.

Remarque Il est important de bien définir les états observables du système parallèle pour lesquels une propriété devra être vérifiée. On fixe ainsi la granularité et le niveau d'atomicité logique des transitions de chaque processus composants.

2.3 Conclusion

Les deux exemples précédents ont montré que dans les deux cas d'interactions (compétition et coopération), un contrôle de l'entrelacement des opérations sur les variables partagées était nécessaire. Mieux, le problème de base est de même nature : comment garantir l'exclusion mutuelle entre différentes sections critiques ? Ce problème de l'exclusion mutuelle est donc le problème de base qu'il faut résoudre pour pouvoir espérer développer des systèmes parallèles dont les exécutions soient sémantiquement correctes. Nous allons donc spécifier puis donner quelques solutions à ce problème sous différentes hypothèses matérielles.

3 Aspects méthodologiques

Tout problème de synchronisation se présente sous la forme d'un énoncé plus ou moins complet. Une première étape est donc de faire une modélisation du problème. Pour ce faire, les systèmes de transitions apportent une solution simple. Chaque processus peut être décrit en terme de transitions significatives vis-à-vis du problème traité. Cette modélisation permet d'abstraire les états "clés" par lesquels passera le processus. Cette étape permet aussi de préciser les variables d'états qui seront partagées par les processus, les conditions logiques de blocage de leur exécution, les problèmes d'ordonnancement des opérations entre processus concurrents, etc.

Une deuxième étape consiste à s'appuyer sur la modélisation précédente pour spécifier plus formellement le problème. On distingue d'une part les propriétés de sûreté qui spécifient le domaine des états corrects du

système et d'autre part les propriétés de vivacité qui spécifient les transitions, opérations qui devront être exécutées. Une catégorie particulière de propriétés de vivacité, sont les propriétés d'équité qui doivent décrire des contraintes d'ordonnancement des actions des processus concurrents.

De telles propriétés peuvent s'exprimer formellement en logique temporelle. Les propriétés de sûreté prennent la forme d'invariants ou de propriétés stables. Les propriétés de vivacité prennent, quant à elles, la forme de prédicats temporels de type « conduit à » (leads to) exprimant des transitions d'un domaine d'états de départ à un domaine d'états d'arrivée. Si le système atteint un état du domaine de départ, alors la propriété spécifie qu'il faudra qu'une suite de transitions conduisent le système dans le domaine d'arrivée. À titre d'exemple de telle propriété, très générique, toute requête d'un processus pour obtenir une ressource doit finalement être satisfaite.

À partir de cette spécification, il faut ensuite développer une solution. Le point le plus délicat est alors de prouver que la solution proposée satisfait bien les propriétés spécifiées. En la matière, dans le cas des systèmes parallèles, le test s'avère souvent très aléatoire, long et complexe compte tenu de l'indéterminisme inhérent aux exécutions parallèles. En conséquence, le raisonnement logique et la preuve de programme restent alors les seuls moyens d'analyse de la correction de tels algorithmes.

4 Le problème de l'exclusion mutuelle

On considère un ensemble de N processus $< P_i >_{0..N-1}$ qui peuvent demander l'accès à une ressource critique. Le droit d'accès à cette ressource sera contrôlé par le respect du protocole suivant :

```
 ...
Entrer(i) ;
/* accès en exclusion mutuelle à la ressource */
Sortir(i) ;
...
```

Les deux procédures Entrer(pid i) et Sortir(pid i) ont donc pour objet de contrôler l'obtention et la libération du droit d'accès en exclusion mutuelle par le processus i.

4.1 Modélisation du problème

Le comportement des processus peut être abstrait pour ne considérer que les transitions d'état significatives vis-à-vis du problème. Tout processus peut ainsi être abstrait sous la forme d'un système de transitions à trois états illustré par la figure 1 :

- État « Hors exclusion » : $P_i.Etat = Hors$, le processus i exécute un traitement ne nécessitant pas l'entrée en exclusion mutuelle ;
- État « Demandeur » : $P_i.Etat = Dem$, le processus i demande à entrer en exclusion mutelle mais n'a pas encore obtenu satisfaction ;
- État « En Exclusion » : $P_i.Etat = Excl$, le processus i a obtenu le droit d'entrée et s'exécute « en exclusion mutuelle »

Notation On notera par $P_i.X$, le prédicat $P_i.Etat = X$.

On remarquera que, dans cette modélisation, l'opération Entrer n'est pas atomique alors que l'opération Sortir l'est. En effet, l'opération Entrer peut nécessiter un blocage logique du processus appelant tant que la condition logique d'entrée en exclusion mutuelle n'est pas satisfaite pour le processus demandeur. La transition vers l'état $P_i.Dem$ marque donc le début de l'opération Entrer. Le processus demandeur reste dans cet état tant que l'opération Entrer n'est pas terminée. C'est la terminaison de l'opération Entrer qui provoquera effectivement la transition vers l'état $P_i.Excl$.

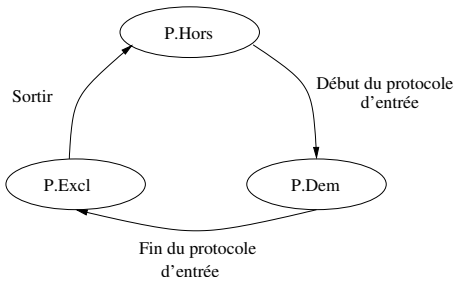

FIG. 1 – Graphe de transitions d'un processus

4.2 Spécification du problème

Une solution correcte au problème doit satisfaire les propriétés suivantes :

– Exclusion mutuelle (Sûreté) : Au plus un processus peut être en exclusion mutuelle, ce qui peut se traduire par l'invariant :

$$\textbf{invariant} \ \ \forall\, i,j \ : \ 0..N-1 \ :: \ P_i.Excl \wedge P_j.Excl \Rightarrow i = j$$

– Absence de famine (Vivacité) : Tout processus demandeur finira par entrer en exclusion mutuelle[1] :

$$\forall i \ : \ 0..N-1 \ :: \ P_i.Dem \ \mapsto \ P_i.Excl$$

Spécification complémentaire Les deux propriétés précédentes suffisent à spéficier toute solution correcte au problème. Cependant, il est intéressant, pour aider à la recherche de solutions correctes, de remarquer que ces spécifications imposent en particulier deux propriétés plus fines sur le comportement des processus : l'une porte sur l'absence d'interblocage et l'autre sur l'indépendance des processus.

– Absence d'interblocage (Sûreté) : Une situation d'interblocage existe lorsque plusieurs processus attendent, pour progresser, une condition logique qui ne deviendra jamais vraie. De façon plus formelle, la condition d'interblocage, dans le cas de l'exclusion mutuelle, peut s'exprimer ainsi :
  – Pour entrer en exclusion mutuelle, tout processus i reste dans l'état demandeur tant qu'une condition G_i n'est pas satisfaite ; ceci peut s'exprimer par :

$$\forall i \ : \ \ P_i.Dem \ \ \textbf{unless} \ \ G_i$$

  – Un risque d'interblocage existe pour un sous-ensemble de processus $I = \{i_0, i_1, \ldots, i_r\}$ d'au moins 2 processus ($|I| \geq 2$) si la propriété stable suivante est vérifiée :

$$\textbf{stable} \ \ \forall k \in I \ : \ \neg G_k$$

  Il y aura effectivement interblocage si la condition stable devient vraie, c'est-à-dire si :

$$true \ \ \mapsto \ \ (\forall k \in I \ : \ \neg G_k)$$

  Les processus de l'ensemble I resteront alors toujours dans l'état demandeur, par suite de la relation :

$$\frac{\textbf{stable} \ \ \forall k \in I \ : \ \neg G_k}{\textbf{stable} \ \ \forall k \in I \ : \ P_k.Dem}$$

– Indépendance (Vivacité) : Si l'accès en exclusion mutuelle est libre et qu'un seul processus est demandeur, celui-ci doit pouvoir entrer en exclusion mutuelle.

$$(\forall i \neq i_0 \ : \ P_i.Hors) \ \wedge \ P_{i_0}.Dem \ \mapsto \ P_{i_0}.Excl$$

[1] Une équité plus forte peut être spécifiée. Par exemple, les processus exécuteront cette transition de demandeur à en exclusion dans l'ordre chronologique de leur demande.

4.3 Recherche de solutions

Bien que le problème paraisse simple, il s'est révélé être délicat à résoudre. Nous envisageons quelques solutions fausses pour mettre en évidence la difficulté du problème avant de présenter des solutions correctes sous différentes hypothèses.

Rappelons une hypothèse et une contrainte que l'on se fixe :

- Une solution est recherchée dans un modèle d'exécution où seules les instructions indivisibles (atomiques) sont la lecture et l'écriture d'un mot mémoire (d'une variable élémentaire).
- On cherche un algorithme symétrique : tous les processus exécutent le même code et il n'existe pas de code « caché » exécuté par un $(N+1)$-ième processus « superviseur », « ordonnanceur ».

Une idée simple : utiliser d'un booléen

L'idée la plus immédiate pour résoudre le problème est d'utiliser une variable partagée booléenne permettant de savoir si un processus est en exclusion mutuelle. Le test de ce booléen devrait permettre de décider si l'opération Entrer peut accorder le droit d'accès au processus appelant. Les procédures d'entrée et de sortie seraient donc les suivantes, dans lesquelles les commentaires précisent les préconditions d'appel et postconditions de sortie ainsi que les transitions d'état.

```
boolean Libre = true ;
```

```
/* Pré :  P_i.Hors */
Entrer( pid i) {
  /*  P_i.Hors  →  P_i.Dem  */
  while ( ! Libre ) {} ;
  Libre = false ;
  /*  P_i.Dem  →  P_i.Excl  */
  }
/* Post :  P_i.Excl  */
```

```
/* Pré :  P_i.Excl  */
Sortir(pid i) {
   Libre = true ;
  }
/* Post :   P_i.Hors  */
```

Malheureusement, cette solution est fausse puisqu'elle n'assure pas la propriété même d'exclusion mutuelle. En effet, un premier processus i_0 exécutant Entrer peut tester la variable Libre et la trouver à vrai. Avant que ce processus poursuive son exécution et affecte la valeur fausse à la variable partagée Libre[2], un autre processus i_1 peut exécuter le début de la procédure Entrer et trouver lui aussi la valeur de Libre à vrai (puisqu'elle n'a pas encore été affectée par le processus i_0). Quel que soit l'ordonnancement ultérieur des processus, ils termineront tous les deux l'exécution de la procédure Entrer. Par conséquent, cette séquence possible d'exécution conduit à un état du système vérifiant :

$$P_{i_0}.Excl \wedge P_{i_1}.Excl \wedge i_0 \neq i_1$$

En fait, un nombre quelconque de processus peuvent être dans l'état $P_i.Excl$ et l'invariant d'exclusion mutuelle n'est pas vérifié.

Une autre idée : Restreindre l'accès aux variables partagées

On peut penser que la solution fausse précédente ne résoud rien parce que la variable booléenne Libre est testée (lue) et mise à jour par les deux processus. Le couplage par variable partagée pourrait être moins fort en restreignant l'accès à une variable partagée de la façon suivante :

- Pas de partage en écriture : un seul processus peut écrire ;
- Partage en lecture : tous les processus peuvent lire la variable.

[2]Il peut être par exemple interrompu pour cause de partage de temps du processeur par quantum.

Pour respecter ce protocole de partage, on remplace la variable d'état portant sur la présence ou non d'un processus en exclusion par des variables d'état mémorisant l'état courant de chaque processus.

On introduit donc un tableau de booléens EnAccès. Pour un processus i, l'élément EnAccès[i] est vrai si le processus est demandeur ou en exclusion. Autrement dit EnAccès[i] $\equiv P_i.Dem \vee P_i.Excl$.

```
boolean EnAccès [N] = false,... ;

Entrer(pid i) {
  boolean seul ;
  EnAccès[i] = true ;
  do {
    seul = true ;
    for (int k=0 ; k<N ; k++)
       if ( k != i ) seul = (seul && !EnAccès[k]) ;
    }
  until seul ;
  /* Pas d'autre processus en accès */
  }
```

```
Sortir(pid i) {
     EnAccès[i] = false ;
     }
```

Cette solution assure la propriété d'exclusion mutuelle. L'argument de base est qu'un seul processus peut trouver que tous les autres ont leur variable EnAccès[.] fausse, puisqu'avant de tester cette assertion, le processus demandeur a déjà positionné sa propre variable à vrai.

Cependant, l'absence d'interblocage [3] n'est pas vérifiée. En effet, en cas de conflit entre seulement deux processus i_0 et i_1, il est possible que les deux processus débutent l'opération Entrer en positionnant à vrai leur variable respective EnAccès[.] avant d'entamer la boucle while. Chacun d'eux vérifie la propriété :

$$P_{i_0}.Dem \quad \textbf{unless} \quad seul_{i_0}$$

$$P_{i_1}.Dem \quad \textbf{unless} \quad seul_{i_1}$$

Aucun ne trouvera alors la condition permettant de sortir de la boucle, c'est-à-dire la variable locale seul de chacun des deux processus ne sera jamais vraie. On entrera alors dans un état stable :

$$\textbf{stable} \quad \neg seul_{i_0} \wedge \neg seul_{i_1}$$

Les deux appels de l'opération Entrer ne se termineront jamais et par conséquent :

$$\textbf{stable} \quad \neg seul_{i_0} \wedge \neg seul_{i_1} \quad \Rightarrow \quad \textbf{stable} \quad P_{i_0}.Dem \wedge P_{i_1}.Dem$$

Une dernière tentative : une variable fixe le gagnant...

Une dernière idée consiste à utiliser le fait qu'une variable n'a qu'une seule valeur à tout instant. Le test de cette valeur peut permettre de « fixer », parmi les processus demandeurs, qui peut entrer en exclusion.

```
pid Tour = 0 ;

Entrer( pid i) {
  while ( i != Tour ) {}
  }
```

```
Sortir(pid i) {
  Tour = (Tour + 1) mod N ;
  }
```

[3] On qualifie ce genre d'interblocage de « livelock », les deux processus étant actifs indéfiniment.

Cette solution simple assure les deux propriétés de sûreté de toute évidence. Un seul processus trouve la valeur qui lui permet de terminer l'opération Entrer et il n'y a pas d'interblocage possible si tous les processus finissent par demander à entrer en exclusion.

Par contre, la propriété d'indépendance n'est pas garantie. Si la variable Tour vaut la valeur i_0, un processus i_1 ne pourra pas entrer en exclusion mutuelle, même si tous les autres processus ne sont pas demandeurs. Bien au contraire, il faudra qu'une « chaîne » de requêtes d'entrée et de sortie partant du processus i_0 finissent par affecter à la variable Tour la valeur i_1.

On constate donc que le problème n'est pas aussi simple qu'il y paraît sous les hypothèses envisagées. En fait, il existe néanmoins des solutions théoriques et pratiques.

4.4 Les solutions « théoriques »

4.4.1 Pour deux processus

Nous présentons ici la solution due à G. L. Peterson[Pet81], trouvée seulement dans les années 80, alors que le problème était connu depuis plus de 20 ans et résolu sous la forme d'un algorithme beaucoup moins élégant de T. Dekker (une description détaillée de cet algorithme est donnée dans [BA82]).

```
type pid = 0..1 ;              pid Tour = 0 ;

Entrer( pid i) {                                    Sortir(pid i) {
  EnAccès[i] = true ;                                  EnAccès[i] = false ;
  Tour = 1-i ;                                         }
  while (EnAccès[1-i] &&( 1-i = Tour ) {}
  }
```

En cas d'accès simultané conflictuel, la valeur de Tour permet de d'ecider quel processus entre en exclusion évitant ainsi l'interblocage.

4.4.2 Pour $N \geq 2$ processus

Des solutions existent pour $N > 2$ processus. La première fut proposée par Dijkstra [Dij65]. Une solution équitable et intéressante par son approche est celle de Lamport[Lam74] appelée algorithme de la boulangerie (Bakery Algorithm). Chaque processus obtient un numéro de passage et le possesseur du plus petit numéro est celui qui peut entrer en exclusion mutuelle. Ce principe est très simple. Le comportement rappelle celui de clients qui prennent un numéro d'ordre pour se faire servir dans un magasin. Le problème est que chaque processus ait un « numéro » distinct des autres demandeurs. Or, nous avons vu que plusieurs processus peuvent lire la même valeur dans une variable. Il faut donc faire appel à une numérotation globale constituée de couples $(num[i], i)$ dans lesquels $num[i]$ est le numéro obtenu par le processus et i son indice (identification). C'est la comparaison de tels couples qui permettra d'ordonner totalement les processus demandeurs. En effet, on définit la relation d'ordre suivante : soit $tour_i = (num[i], i)$ et $tour_j = (num[j], j)$, alors la relation de précédence notée $\prec$ entre ces deux couples (dates) est définie par :

$$tour_i \prec tour_j \;\equiv\; num[i] < num[j] \vee (num[i] = num[j] \wedge i < j)$$

On constate que si deux processus ont obtenu le même numéro, leur indice les départage.

Une difficulté de l'algorithme réside cependant dans le test même du prédicat « avoir le plus petit couple $(num[i], i)$ ». Plus précisément, ce prédicat a la forme suivante pour un processus i_0 :

$$\forall i : 0..N-1 \wedge i \neq i_0 \;::\; tour_{i_0} \prec tour_i$$

C'est l'introduction du tableau EnAcq qui permet d'éviter une mauvaise évaluation de ce prédicat en distinguant une phase d'acquisition du numéro pour chaque processus.

```
type pid = 0..N-1 ;
boolean EnAcq[N] ;  int num[N] ;

Entrer( pid i) {
  /* Phase d'acquisition d'un numéro */
  EnAcq[i] = true ;
  int tour = 0 ;for (k=0 ; k<N ; k++) tour = Max(tour, num[k]) ; num[i] = tour + 1 ;
  EnAcq[i]=false ;
  /* Phase d'attente éventuelle de son tour */
  for (k=0 ; k<N ; k++) {
     while (EnAcq[k]) do {} ;
     while (num[k] != 0 && (num[k],k) ≺ (num[i],i)) {}
     }
  }

Sortir(pid i) {  num[i] = 0 ; }
```

4.5 Les solutions « pratiques »

Dans la pratique, deux solutions sont utilisées selon que l'architecture sous-jacente est mono ou multi-processeur.

Le cas monoprocesseur

Dans le cas monoprocesseur, une solution simple existe. En effet, le problème de base, nous l'avons vu, est de disposer d'une opération atomique de lecture-écriture d'un mot mémoire. Il ne faut pas que des lectures par d'autres processus s'intercalent entre la lecture et l'écriture faite par un processus donné. Or, avec une architecture monoprocesseur, pour éviter tout entrelacement, il suffit de passer en mode d'exécution ininterruptible. Puisqu'un seul processeur existe, il n'y a qu'un seul processus actif et il suffit qu'il puisse garder le contrôle du processeur. Dans ce cas, l'indivisibilité de la lecture-écriture peut être obtenue par l'ininterruptibilité.

Une solution simple peut être décrite de la façon suivante :

```
                    boolean Libre = true ;

Entrer( pid i) {                                          Sortir(pid i) {
  IT_Masquer() ;                                            Libre = true ;
  while (!Libre) {                                          }
     /* Fenêtre interruptible */
     IT_Démasquer() ; commuter() ; IT_Masquer() ;
     }
  Libre = false ;
  IT_Démasquer() ;
  }
```

On remarquera l'obligation de ne pas rester dans une boucle de test ininterruptible lorsque la variable Libre est fausse. En effet, cette boucle serait infinie puisqu'aucun autre processus (en particulier celui en exclusion mutuelle) ne pourrait plus prendre le contrôle du processeur unique pour s'exécuter et finalement permettre au processus en exclusion d'exécuter l'opération Sortir.

Le cas multiprocesseur

Dans le cas multiprocesseur, la simplification du problème est apportée par le matériel. En réalité, l'hypothèse d'une lecture atomique et d'une écriture atomique n'est aujourd'hui plus guère assurée dans la plupart des multi-processeurs. La présence de caches peut très bien ne pas garantir la lecture de la dernière valeur écrite lors de l'accès à une variable partagée. Par conséquent, il est indispensable de disposer dans ce cas d'instructions spécifiques.

Les multiprocesseurs disposent par exemple d'une instruction de lecture-écriture atomique, qui prend la forme d'une instruction dite « test and set ». Cette instruction peut se réduire à la lecture du bit de signe (test $< 0, = 0, > 0$) et à l'écriture d'un 1 dans ce bit (le contenu du mot devient donc négatif).

Avec une telle instruction, l'utilisation d'une variable partagée permet de résoudre le problème. Le processus demandeur doit lire une valeur positive ou nulle pour pouvoir entrer en exclusion mutuelle. Dès que la valeur a été lue, la variable est devenue négative. Par conséquent aucun autre processus ne pourra lire une valeur positive ou nulle jusqu'à une nouvelle affectation de la variable.

Si l'on note une telle instruction int TAS(int x), la solution générale au problème de l'exclusion mutuelle peut prendre la forme simple suivante :

```
                        int Libre = 0 ;
Entrer( pid i) {                              Sortir(pid i) {
  while (TAS(Libre)<0) {}                       Libre = 0 ;
  }                                             }
```

Cette solution est faiblement équitable. En fait, il est difficile de donner une « preuve » explicite de l'équité de cette solution. En effet, dans l'absolu, un processus « malchanceux » pourrait toujours exécuter l'instruction TAS au moment où la variable libre est négative. Cependant, en pratique, la probabilité que plusieurs processus arrivent à se synchroniser pour « voler » l'accès à un processus donné est nulle.

4.6 Conclusion

Le problème de l'exclusion trouve dans la pratique une solution simple. Cependant, la recherche d'une solution sous des hypothèses faibles montre la difficulté de construire des solutions correctes aux problèmes de synchronisation. La difficulté des solutions tient essentiellement à l'explosion combinatoire des états possibles du système parallèle de processus dû au non déterminisme des exécutions sous les hypothèses asynchrones.

Nous remarquerons aussi l'importance de la modélisation du problème et de sa spécification en terme de propriétés de sûreté et de vivacité (y compris équité).

5 Concept de sémaphore

Le concept de sémaphore constitue le premier mécanisme de synchronisation pour contrôler l'ordonnancement de processus parallèles proposé par E. W. Dijkstra [Dij68]. Ce mécanisme, comme tous les concepts qui seront proposés ultérieurement, permet essentiellement de s'abstraire de la gestion explicite de files d'attente de processus bloqués intervenant dans l'ordonnancement à moyen terme des processus.

L'idée directrice est d'encapsuler une variable partagée de type compteur positif ou nul. Le mécanisme proposé a pour objectif d'assurer l'exécution cohérente des opérations de test, d'incrémentation et de décrémentation d'un tel compteur accédé par plusieurs processus. Le mécanisme doit donc contrôler l'ordonnancement des appels d'opérations conflictuelles sur la variable compteur partagée. Il faudra, dans certains cas, retarder l'exécution d'une opération et donc bloquer momentanément le processus appelant.

5.1 Spécification du mécanisme

Soit S un objet de type sémaphore et $S.Cnt$ le compteur qu'il encapsule. Le mécanisme de sémaphore garantit un premier invariant : le compteur restera positif ou nul $S.Cnt \geq 0$. De plus, seules deux primitives

P et V permettent de modifier ce compteur. Ces deux primitives sont (doivent être) **atomiques** et leur sémantique sous la forme de triplets de Hoare est la suivante :

$$\begin{array}{ccc} \{S.Cnt = k \wedge k > 0\} & P(S) & \{S.Cnt = k - 1\} \\ \{S.Cnt = k\} & V(S) & \{S.Cnt = k + 1\} \end{array}$$

Autrement dit, la primitive P assure une décrémentation atomique du compteur à condition que celui-ci soit positif. Cette dernière condition peut entraîner l'impossibilité d'une exécution immédiate de la primitive. Elle constitue donc une condition dite de blocage. Autrement dit, un appel de la primitive ne sera pas forcément suivi d'une exécution immédiate. Celle-ci pourra être retardée jusqu'au moment où la condition devient vraie. La primitive V assure, quant à elle, une incrémentation atomique et reste toujours exécutable de façon immédiate.

5.2 Propriétés fondamentales

- Si la précondition $Pre(P(S) = (S.Cnt > 0)$ n'est pas vérifiée, le processus appelant est bloqué. L'opération est donc retardée.
- Toute opération V rend la précondition précédente vraie en incrémentant le compteur. En conséquence, s'il existe au moins un processus bloqué sur l'appel de la primitive P, alors un tel processus et un **seul** est débloqué (le processus passe dans l'état prêt ou même actif).
- les règles d'ordonnancement précédentes assurent l'invariant suivant, dans lequel $\#op$ représente le nombre d'opérations *op* exécutées depuis la création de l'objet sémaphore :

$$\textbf{invariant} \quad S.Cnt = S.Cnt0 + \#V - \#P$$

Une façon simple d'interpréter la sémantique d'un s'emaphore est de la considérer comme un distributeur de tickets. Exécuter une operation P, c'est prendre un ticket. Ex'ecuter une operation V s'est restituer ce ticket. Le sémaphore compte les tickets disponibles. Lorsque le compteur est nul, il n'y a pas de ticket disponible, et les opération P's doivent être retardées.

On remarquera qu'un sémaphore permet un schéma d'interaction entre deux processus, le processus « réveilleur » et le processus « réveillé » s'il existe.

Un problème se pose dans le choix du processus réveillé lorsque plusieurs processus sont candidats au réveil. Une stratégie simple consiste à ce que l'ordre de réveil soit identique à l'ordre chronologique de blocage des processus sur le sémaphore. Tout autre stratégie (priorité entre processus par exemple) peut conduire à un phénomène de famine de certains processus.

5.3 Utilisation d'un sémaphore

Le mécanisme de sémaphore permet de résoudre simplement de nombreux schémas, modèles génériques de synchronisation. À titre d'exemple, on peut décrire une solution simple au problème de l'exclusion mutuelle ou de l'allocation de ressources critiques.

Solution au problème de l'exclusion mutuelle

Rappelons que ce problème consiste à garantir qu'un seul processus au plus exécute une section critique. Pour exécuter une telle section, tout processus respecte le protocole suivant :

```
Entrer() ;
/* code de la section critique */
Sortir() ;
```

Le contrôle de la présence d'un processus en exclusion mutuelle peut être mémorisé par une simple variable booléenne *Libre* initialisée à vrai. La sémantique des opérations Entrer() et Sortir() peut alors s'exprimer sous forme de triplets de Hoare par :

$$\{Libre\} \;\; Entrer() \;\; \{\neg Libre\} \qquad\qquad \{\neg Libre\} \;\; Sortir() \;\; \{Libre\}$$

Cependant, nous avons vu que la mise-à-jour de la variable *Libre* par plusieurs processus n'assure pas la propriété d'exclusion mutuelle. Les processus qui entrent et celui qui sort d'exclusion testent et/ou mettent à jour cette variable partagée. Pour obtenir une solution correcte, il suffit cependant d'encapsuler cette variable dans un sémaphore.

On utilise un seul sémaphore initialisé à la valeur 1 (vrai) et toute séquence de code devant s'exécuter en exclusion mutuelle est parenthésée par l'appel de P (parenthèse ouvrante) et V (parenthèse fermante).

```
sémaphore mutex init 1 ;
...
P(mutex) ;
/* code de la section critique */
V(mutex) ;
```

Dans ce cas particulier d'usage, l'entier encapsulé dans le sémaphore ne peut prendre que les valeurs 0 ou 1. On peut donc obtenir le même mécanisme en encapsulant un simple booléen B initialement vrai avec la sémantique suivante :

$$\begin{array}{ccc} \{S.B\} & P(S) & \{\neg S.B\} \\ \{true\} & V(S) & \{S.B\} \end{array}$$

Ce type de sémaphore est appelé souvent sémaphore d'exclusion mutuelle tant son usage est courant dans ce cas de figure.

Solution au problème de l'allocation de ressources critiques

On suppose qu'il existe N ressources critiques (imprimantes, scanners, tampons mémoire par exemple). Pour assurer l'usage exclusif de ces ressources, il faut résoudre le problème du comptage correct de ces ressources. Pour cela, on peut utiliser un sémaphore encapsulant ce compteur. Pour utiliser une des ressources disponibles, un processus doit respecter le protocole classique suivant :

```
uneR = Allouer() ;
/* utiliser uneR */
Libérer(uneR) ;
```

Les opérations Allouer et Libérer doivent garantir l'usage exclusif de la ressource référencée :

$$\textbf{invariant} \quad P_i.utilise(r) \wedge P_j.utilise(r) \quad \Rightarrow \quad i == j$$

L'exécution de l'opération Allouer est conditionnée par la précondition « il existe au moins une ressource disponible » puisqu'elle entraînera la décrémentation de 1 du nombre de ressources disponibles. L'opération de libération s'accompagnera de façon symétrique d'une incrémentation de 1 du nombre de ressources disponibles. Si la variable nL compte le nombre de ressources disponibles, la sémantique des opérations Allouer et Libérer s'exprime en logique de Hoare par :

$$\begin{array}{ccc} \{nL = k \wedge k > 0\} & Allouer() & \{nL = k - 1\} \\ \{nL = k\} & Liberer(\ldots) & \{nL = k + 1\} \end{array}$$

On constate que cette variable nL est typiquement une variable partagée modifiée (testée, incrémentée et décrémentée) par plusieurs processus. Par conséquent, ces modifications ne seront cohérentes que si elles sont exécutées en exclusion mutuelle. Le sémaphore fournit le mécanisme adéquat pour encapsuler cette variable. On utilise donc un sémaphore SL, compteur de ressources critiques, initialisé à N. On peut récrire la sémantique des opérations sous la forme :

$$\begin{array}{ccc} \{SL.Cnt = k \wedge k > 0\} & P(SL); Allouer() & \{SL.Cnt = k - 1\} \\ \{SL.Cnt = k\} & Liberer(\ldots); V(SL) & \{SL.Cnt = k + 1\} \end{array}$$

dans lesquels on a simplement substitué la variable abstraite nL par la variable entière du sémaphore $SL.Cnt$ et introduit les opérations sur le sémaphore.

5.4 Implantation

La notion de sémaphore peut être implantée de multiple manières. Cependant, une implantation intéressante impose d'assurer un blocage passif du processus appelant une opération P si la précondition n'est pas satisfaite.

Nous donnons une implantation sous la forme d'une classe Java en supposant l'existence d'une classe Processus implantant le concept de processus et d'une classe Fifo implantant la notion de file d'attente avec les opérations d'insertion et extraction respectant l'ordre chronologique. La classe Processus implante la primitive Suspendre() qui arrête le processus (actif) appelant et donne le contrôle du processeur à un autre processus prêt (s'il en existe un) et la primitive Continuer qui replace le processus opérande parmi les processus exécutables. Par ailleurs, on suppose accessible le processus actif exécutant une opération sur le sémaphore via une primitive getActif.

Il reste à garantir l'atomicité des opérations P et V. Pour cela, il suffit de considérer les opérations P et V comme des opérations devant s'exécuter en exclusion mutuelle. C'est pourquoi, on trouve en commentaire en début et fin des opérations le protocole d'entrée et sortie d'exclusion mutuelle. Reste à trouver une solution pour ce protocole parmi celles du chapitre précédent. Dans le cas monoprocesseur, on utilise le mode ininterruptible par masquage/démasquage des interruptions. Dans le cas multi processeurs, on a recours à l'instruction TAS sur un verrou.

```
class  Sémaphore {
   /* représentation d'un sémaphore */
      int  Cnt ;
      Fifo bloqués ;
   P() {
      /* entrer en exclusion mutuelle ... */
      Cnt-- ;
      if ( Cnt < 0) {
          bloqués.Insérer(Processus.getActif()) ; Processus.Suspendre() ;
          }
      /* sortir d'exclusion mutuelle ... */
   }
   V() {
      /* entrer en exclusion mutuelle ... */
      Cnt++ ;
      if ( Cnt  ≤  0) {
          Processus réveillé = (Processus) bloqués.Extraire() ;
          réveillé.Continuer() ;
          }
      /* sortir d'exclusion mutuelle ... */
   }
   Sémaphore( int v0) {
      if ( v0 < 0 ) throw new Error("ValeurErronée") ;
      Cnt = v0 ; bloqués = new Fifo() ;
   }
}
```

Remarque Dans cette implantation, le compteur du sémaphore peut devenir négatif. En fait, lorsque le compteur devient négatif, sa valeur absolue représente le nombre exact de processus bloqués dans la file d'attente du sémaphore. Ceci permet le test de l'existence d'au moins un processus bloqué, introduit dans l'opération V.

5.5 Conclusion

Les sémaphores permettent de résoudre les problèmes de synchronisation de type conflictuel ou coopératif. Leur usage à travers les exemples présentés semble relativement simple et sûr. Cependant, tous les problèmes de synchronisation ne sont pas aussi facilement résolus. En effet, la condition de blocage associée au sémaphore est fixe : valeur nulle d'un compteur. Or, toutes les conditions logiques de blocage rencontrées dans les problèmes de synchronisation ne s'expriment pas simplement sous cette forme. Par ailleurs, le mécanisme de sémaphore établit une relation de « réveilleur » à « réveillé » entre seulement deux processus (relation 1-1). Nous verrons que certains problèmes sont plus simplement résolus avec un mécanisme permettant le reveil de plusieurs processus par un seul processus réveilleur (relation 1-N). C'est pourquoi, un mécanisme plus général a été proposé, en l'occurrence le mécanisme de moniteur.

6 Concept de moniteur

Le concept de moniteur vise à pallier plusieurs inconvénients des sémaphores. En effet, les sémaphores sont des objets de synchronisation d'un type prédéfini assez primitif. Bien qu'ils constituent une brique de base permettant de résoudre tous les problèmes de synchronisation dans un contexte centralisé, la résolution par sémaphore d'un problème de synchronisation est souvent délicate. De plus, l'expression de la synchronisation est mélangée au code des opérations synchronisées.

La notion de moniteur vise à corriger ces insuffisances sous la forme d'un constructeur de mécanismes de synchronisation. Un moniteur permet de décrire une solution à un problème particulier de synchronisation sous la forme d'un module partagé.

Nous présentons dans ce qui suit la notion de moniteur telle qu'elle a été définie par C.A.R Hoare [Hoa74]. Cette notion a été aussi proposée par Brinch Hansen [BH72, BH75]. Nous présentons ensuite quelques variantes proposées ainsi que les « formes approchées » du mécanisme de moniteur que l'on peut trouver soit dans les noyaux de systèmes (POSIX par exemple) soit dans les langages de programmation (Java par exemple).

6.1 Spécification du mécanisme

Un moniteur de Hoare est un module partagé et partageable par plusieurs processus. En tant que module, il en possède toutes les propriétés de visibilité des objets définis dans un module : les constantes, types et variables définis dans le module sont invisibles à l'extérieur du module (masquage d'information) et seul, un ensemble de procédures constituant l'interface du module sont appelables par les processus. Ces procédures ne peuvent accéder qu'aux paramètres effectifs fournis lors des appels et aux variables locales au (déclarées dans le) moniteur. Par rapport au mécanisme de sémaphore, on a une généralisation du mécanisme d'encapsulation : un sémaphore encapsule toujours une simple variable partagée entière ou booléenne alors que le module, en fournissant le même principe d'encapsulation de variables quelconques le généralise.

Syntaxe de définition

La structure syntaxique d'un moniteur est donc identique à celle d'un module ou d'une classe. Le mot clé module (ou class) est remplacé par le mot clé moniteur.

```
moniteur < id. du moniteur > {
   < déclarations locales de constantes, types, variables et procédures >
   public < id. de procédure 1 > (...) { < corps de la procédure 1 > }
   ...
   public < id. de procédure N > (...) { < corps de la procédure N > }
   /* constructeur */
   < id. du moniteur > (...) { < initialisation d'un objet moniteur > }
}
```

Tout appel de procédure publique est préfixé par la référence à l'objet moniteur selon la syntaxe habituelle :

```
< référence moniteur >.< id. de procédure (publique) > (...)
```

6.2 Propriété fondamentale : Criticité d'un moniteur

Pour garantir une gestion cohérente des variables partagées encapsulées dans le moniteur, tout moniteur est considéré comme une ressource critique. Autrement dit, à tout moment, un seul processus au plus peut obtenir le droit d'usage du moniteur pour exécuter tout ou partie d'une de ses procédures publiques. En conséquence, tout appel de procédure d'un moniteur peut conduire à un blocage du processus appelant si un autre processus a déjà obtenu le droit d'accès à la ressource critique moniteur via un autre appel de procédure. L'accès au moniteur sera libéré par le processus actif dans le moniteur lors de la terminaison de l'exécution de sa procédure[4]

Grâce à cette propriété de criticité, un invariant pourra être défini sur les variables locales aux moniteurs. Cet invariant devra être vérifié lorsque le moniteur est libre (pas d'opération en cours).

6.3 Traitement des conditions logiques de synchronisation

L'exclusion mutuelle ne suffit pas pour résoudre les problèmes de synchronisation. D'autres conditions de blocage apparaissent dans les problèmes. Par conséquent, le mécanisme de moniteur offre la possibilité de programmer explicitement le contrôle de conditions logiques d'exécution (de blocage) durant l'exécution des opérations du moniteur. Lorsqu'une condition ne sera pas satisfaite, le processus appelant pourra être mis en file d'attente dans l'état bloqué. Il libérera ainsi l'accès au moniteur au profit d'un autre processus en attente d'usage du moniteur. Pour ce faire, un type prédéfini appelé condition est introduit. Des variables de ce type ne peuvent être déclarées que dans un moniteur. Ce type offre deux opérations : wait et signal. Soit C une variable de type condition déclarée dans un moniteur M, alors :

- l'exécution de l'opération C.wait() dans une opération du moniteur entraîne le blocage du processus appelant dans la file d'attente associée à cette variable. Le processus appelant n'étant plus actif dans le moniteur, l'accès au moniteur est libéré et un autre processus peut accéder au moniteur.
- l'exécution de l'opération C.signal() dans une opération du moniteur entraîne le choix d'un **seul** processus bloqué dans la file de la variable C et le réveil de ce processus. Plus précisement, le processus « signaleur » (ayant appelé l'opération signal) est suspendu. Celui-ci libère donc l'accès au moniteur au profit **immédiat** du processus réveillé qui obtient l'accès au moniteur et peut donc poursuivre l'opération qu'il exécutait dans le moniteur. Cette règle d'ordonnancement est très importante. En effet, une opération signal est exécutée lorsqu'une condition logique attendue par des processus est devenue vraie. Il faut donc être sûr que cette condition restera vraie jusqu'à ce que le processus choisi et réveillé puisse s'exécuter dans le moniteur. Si un autre processus que le réveillé prenait le contrôle du moniteur, il risquerait de rendre fausse la condition avant que le réveillé ne s'exécute. Ce passage du droit d'accès au moniteur entre le signaleur et le réveillé est un des éléments fondamentaux du mécanisme de moniteur.

Exemple

Le concept de moniteur peut permettre d'implanter par exemple le mécanisme de sémaphore. Un type sémaphore peut être représenté par un moniteur Semaphore exportant les deux opérations P et V et offrant un constructeur Semaphore pour instancier des objets. Un objet sémaphore est représenté par un compteur entier positif (cf. invariant) ou nul et une variable condition permettant de bloquer les processus qui tentent d'exécuter une opération P alors que le compteur est nul. On peut constater que la variable condition permet de s'abstraire de (d'éviter) la gestion d'une file d'attente explicite de processus.

[4]Nous allons voir qu'il existe d'autres cas de libération dans ce qui suit.

```
moniteur Semaphore { /* invariant  cnt  ≥  0  */
   int cnt ;
   condition positif ;
   public P() {
     if (cnt == 0) positif.wait() ;
     /* cnt > 0 */
     cnt-- ;
     }
   public V() {
     cnt++ ;
     /* cnt > 0 */
     positif.signal() ;
     }
   Semaphore ( int v0) {
     cnt = v0 ;
   }
}
```

On remarquera la définition de l'invariant du moniteur ainsi que les prédicats associés d'une part à la post-condition d'une opération wait et à la précondition d'une opération signal. Les règles d'ordonnancement de réveil des processus bloqués permettent de vérifier la cohérence des opérations en assurant que pour toutes les variables conditions du moniteur et toute occurrence de couple d'opérations (c.signal, c.wait), l'implication suivante est valide :

$$\forall c, i, j : \textbf{condition} \;\; \forall c.signal^{(i)}, c.wait^{(j)} \;\; :: \;\; Pre(c.signal^{(i)}) \;\Rightarrow\; Post(c.wait^{(j)})$$

Création et utilisation de moniteurs « sémaphores » Après création d'un objet sémaphore de type moniteur par : `Semaphore S = new Semaphore(N)`, on peut invoquer les opérations de façon classique par S.P() et S.V().

Remarque On peut remarquer que la notion de variable condition évite, comme les sémaphores, la gestion explicite de files d'attente de processus bloqués.

6.4 Exemple d'usage des moniteurs

À titre d'exemple, considérons le problème générique de communication entre processus selon un schéma de producteur(s)-consommateur(s) via un tampon mémoire à N blocs.

Le moniteur encapsule un espace mémoire de N blocs dans lesquels peuvent être mémorisés les messages des producteurs. Les consommateurs viendront retirer les messages dans l'ordre chronologique de dépot. On suppose que tout message tient dans un seul bloc. Sous ces hypothèses, une gestion cohérente des blocs mémoires et des messages doit assurer l'invariant :

$$\textbf{invariant} \;\; nV \geq 0 \;\wedge\; nP \geq 0 \;\wedge\; nV + nP = N$$

dans lequel, l'entier nV est égal au nombre de blocs vides et l'entier nP est égal au nombre de blocs pleins. Relativement à ces compteurs, les opérations de dépot et de retrait de message ont la sémantique suivante :

$$\{nV = k \;\wedge\; k > 0 \;\wedge\; nP = N - k\} \; Deposer \; \{nV = k - 1 \;\wedge\; nP = N - (k-1)\}$$

$$\{nP = k \;\wedge\; k > 0 \;\wedge\; nV = N - k\} \; Retirer \; \{nP = k - 1 \;\wedge\; nV = N - (k-1)\}$$

Le maintien de l'invariant impose bien sûr que chaque opération a un effet sur les deux compteurs. Le protocole de communication est décrit par l'interface suivante :

```
moniteur TamponPartagé { /* invariant nV ≥ 0 ∧ nP ≥ 0 ∧ nV + nP = N */
   /*    {nV = k ∧ k > 0 }  */
   public Déposer(Message m) ;
   /*     {nV = k − 1 }   */

   /*  {nP = k ∧ k > 0 }  */
   public Message Retirer() ;
   /*     {nP = k − 1 }   */

   TamponPartagé (int N) ;
}
```

L'implantation du moniteur repose sur l'utilisation de deux variables de type conditions qui permettent de bloquer les producteurs si aucun bloc n'est vide et de bloquer les consommateurs si aucun bloc ne contient de message.

L'ensemble des blocs est géré comme un tampon circulaire de façon à respecter l'ordre chronologique entre dépots et retraits. Le moniteur encapsule donc un tableau de N blocs qui pour simplifier sont déclarés de type Message. Deux indices de tête et queue repèrent respectivement le prochain bloc plein à vider et le prochain bloc vide à remplir [5].

```
moniteur TamponPartagé { /* invariant nV ≥ 0 ∧ nP ≥ 0 ∧ nV + nP = N */
    int nV ; int nP ;
    condition nonPlein ; condition nonVide ;
    int taille ; int t ; int q ; Message[] blocs ;
   public Déposer(Message m) {
      if (nV == 0) nonPlein.wait() ;
      /*  nV > 0  */
      nV-- ; blocs[q] = m ; q = (q + 1) % taille ; nP++ ;
      /*  nP > 0   */
      nonVide.signal() ;
   }
   public Message Retirer() {
      if (nP == 0) nonVide.wait() ;
      /*  nP > 0   */
      nP-- ; Message me = blocs[t] ; t = (t + 1) % taille ; nV++ ;
      /*  nV > 0   */
      nonPlein.signal() ;
      return me ;
   }
   void TamponPartagé( int N ) {
     taille = N ; blocs = new Message[taille] ;
     nV = taille ; nP = 0 ; q = 0 ; t = 0 ;
   }
}
```

6.5 Quelques variantes, extensions

Diverses variantes ou extensions ont été proposées. Elles portent sur deux aspects importants :

– L'extension du mécanisme de réveil : comme le sémaphore, le moniteur de Hoare ne réveille au plus qu'un processus lors d'une opération signal. Ceci soulève quelques difficultés de programmation lorsque certains problèmes comportent le besoin de réveiller plusieurs, voire tous les processus en attente d'une

[5] Le symbole % dénote l'opérateur modulo.

même condition. Par exemple, si un processus libère $P > 1$ ressources critiques, il peut exister plusieurs processus en attente qui peuvent peut-être voir leur requête satisfaite. Un opération signal réveillant tous les processus bloqués (vidant la file d'attente de la variable condition) a donc été proposée. En fait, les processus réveillés doivent cependant reconquérir un à un le droit d'accès au moniteur pour conserver la propriété de criticité du moniteur. Par ailleurs, chaque processus réveillé doit vérifier dès son réveil que la condition qu'il attendait est bien encore satisfaite : d'autres réveillés, ayant obtenu l'usage du moniteur avant lui, ont en effet pu invalider cette condition.

– L'affaiblissement des contraintes d'ordonnancement lors du réveil d'un processus. La priorité du processus réveillé sur le processus signaleur proposée par Hoare apporte une simplification importante pour vérifier la validité d'un moniteur. Cependant, cette contrainte forte peut entrer en conflit avec par exemple des priorités fixées a priori entre les processus. Dans ce cas, si le processus signaleur est plus prioritaire que le processus réveillé, la contrainte d'ordonnancement des moniteurs devient contradictoire avec la priorité d'exécution des processus. C'est pourquoi, dans la pratique, on utilise souvent une forme affaiblie de moniteurs : l'ordonnancement entre signaleur et réveillé n'est pas fixé. Bien évidemment, cela signifie qu'un processus réveillé doit tester explicitement, après son réveil, si la condition logique qu'il attendait est bien toujours vérifiée. Les réveils « inutiles », infructueux sont alors possibles et il faut prendre garde aux risques de famine qui peuvent en résulter.

6.6 Implantation des moniteurs

Le moniteur en tant que module partagé doit être implanté par un ensemble de primitives de gestion de l'ordonnancement des processus. Ces primitives doivent :

– Contrôler le début et la fin de toute opération dans le moniteur pour assurer l'exclusion mutuelle d'utilisation du moniteur. Ceci revient à gérer un verrou d'accès associé au moniteur. Toute opération du moniteur doit alors être parenthésée par le verrouillage/déverrouillage de l'accès au moniteur.

– Implanter les variables conditions. En général, la primitive signal n'assure pas la priorité d'accès au moniteur pour le processus réveillé.

Dans la pratique, on dispose le plus souvent non pas de la structure syntaxique « moniteur » au niveau langage de programmation, mais simplement des primitives d'implantation énoncés ci-dessus. À titre d'exemple, le moniteur sémaphore pourrait être récrit sous la forme d'une simple classe en utilisant de telles primitives :

```
class Semaphore { /* invariant  cnt ≥ 0  */
   int cnt ; verrou  mutex ; condition positif ;
   public P() {
     mutex.lock() ;
     if (cnt == 0) positif.wait( mutex) ;
     /* cnt > 0 */
     cnt-- ;
     mutex.unlock() ;
     }
   public V() {
     mutex.lock() ;
     cnt++ ;
     /* cnt > 0 */ positif.signal() ;
     mutex.unlock() ;
     }
   Semaphore ( int v0) {
     cnt = v0 ;
   }
}
```

On remarquera que la primitive wait possède en paramètre le verrou associé au moniteur pour pouvoir libérer l'accès au moniteur par déverrouillage. Le processus bloqué devra réussir une opération de verrouillage (cachée dans la primitive wait) avant de pouvoir poursuivre son exécution dans le moniteur.

6.7 Conclusion

Les moniteurs ne constituent pas un mécanisme plus puissant que les sémaphores. En fait, il est possible d'implanter le mécanisme de moniteur avec des sémaphores et nous avons vu qu'un moniteur pouvait implanter un sémaphore. Néanmoins, les moniteurs permettent une programmation plus fiable des problèmes de synchronisation grâce en particulier au contrôle de l'invariant et de la cohérence des préconditions de réveil vis-à-vis des postconditions de blocage.

Le concept lui-même n'a été que rarement introduit au niveau syntaxique dans les langages (citons le Pascal parallèle [BH75] par exemple). Cependant, les primitives d'implantation du mécanisme sont aujourd'hui disponibles soit sous forme de bibliothèque soit dans le noyau de gestion de threads (processus légers) de nombreux systèmes d'exploitation. Les idées et principes proposés sont donc bien passés dans la pratique courante de la programmation parallèle.

7 Synchronisation et communication

Les mécanismes précédents d'exclusion mutuelle, de sémaphore, de moniteur, ont été présentés dans un contexte d'exécution centralisé. Les objets de synchronisation sont supposés partagés via une mémoire commune accessible aux processus usagers. Ce partage est le plus souvent réalisé via le noyau d'exécution sous-jacent qui offre les primitives de gestion (création/destruction et usage) des mécanismes proposés.

Un autre modèle d'exécution est aujourd'hui très utilisé : le modèle réparti. Les processus ne s'exécutent plus sur le même ordinateur mais doivent néanmoins se synchroniser pour réaliser les traitements applicatifs. Dans un tel contexte, les processus ne peuvent que s'échanger des messages pour parvenir à se synchroniser. L'hypothèse de l'existence d'une mémoire commune n'est plus vérifiée. Il s'agit donc de synchroniser des processus communicants. Ce modèle peut être étudié et utilisé indépendamment de l'architecture sous-jacente (centralisée ou répartie).

Nous rappelons tout d'abord les principes de base des protocoles de communication par messages. Le caractère élémentaire de la communication par message conduit à définir des protocoles de plus haut niveau facilitant la description de la synchronisation de processus communicants. À titre d'exemple, nous décrivons le mécanisme de rendez-vous proposé dans le langage Ada pour synchroniser les tâches dans les applications critiques et "temps réel". Dans le cadre du langage Ada, la synchronisation par rendez-vous a été utilisée le plus souvent dans un contexte centralisé compte tenu du domaine d'application Temps Réel. Cependant, le modèle proposé relève pleinement des processus communicants. Pour une description complète du langage on peut se reporter au livre de John Barnes [Bar00]. Pour les aspects concernant exclusivement la synchronisation, une présentation complète en est donné dans [Pad90].

7.1 La communication par message

La communication par messages entre processus suppose la définition d'un protocole précis d'échange des messages. De nombreux paramètres interviennent en effet dans une telle communication. Nous considérons ici le type de protocole le plus élémentaire comportant seulement deux opérations émettre et recevoir permettant à deux processus de communiquer selon un protocole de type point à point. Même dans ce cas, de nombreuses variantes existent :

– la synchronisation entre émetteur et récepteur : il s'agit de préciser à quel moment une opération d'émission se termine pour l'émetteur. Si l'émetteur est bloqué jusqu'à ce que le message envoyé ait été reçu et acquitté par le récepteur, on parle de communication **synchrone par rendez-vous**. Sinon, on peut considérer qu'il s'agit d'une communication de type **asynchrone** sachant bien que ce terme désigne alors non pas un mais des protocoles divers : la seule propriété commune étant que l'opération

d'émission se termine sans attendre d'acquittement de la part du récepteur. Dans ce type de protocole, la connaissance de l'état du récepteur par l'émetteur est donc beaucoup plus flou et la programmation en "asynchrone" est donc souvent plus délicate bien qu'elle possède l'avantage d'offrir un plus fort parallélisme potentiel entre émetteur et récepteur.
- La communication entre deux processus nécessite un moyen de désignation direct ou indirect du récepteur. C'est pourquoi, la communication entre processus nécessite le plus souvent une phase préalable de connexion.
- La communication peut être occasionnelle ou établie pour une période donnée, fiable ou non fiable. On parle de datagramme dans le cas occasionnel et non fiable, par exemple le protocole UDP (User Datagram Protocol), ou de circuit virtuel dans le cas fiable, par exemple TCP (Transmission Control Protocol) .

7.2 La synchronisation dans le langage Ada

Le langage Ada est un langage de programmation destiné aux applications critiques embarquées. Ce genre d'applications comporte des contraintes de temps dans l'exécution des traitements et présente un caractère réactif (plutôt que fonctionnel). Classiquement, de telles applications contrôlent, via des actionneurs, un environnement perçu via des capteurs. Les traitements peuvent être cycliques et périodiques (échantillonnage des entrées sur les capteurs) ou déclenchés de façon asynchrone sur l'occurrence d'un événement (alerte par exemple). Enfin, des traitements de fond peuvent être exécutés lorsque du temps processeur reste libre après exécution des traitements plus urgents.

Pour structurer de telles applications parallèles, le concept de tâche a été proposé. Très voisin de celui de processus, une tâche est l'exécution d'un code de façon synchrone (périodique) ou asynchrone avec une priorité d'urgence liée à une échéance au-delà de laquelle l'exécution est considérée comme trop tardive (hors délai). C'est surtout ces contraintes temporelles qui distinguent réellement une tâche d'un processus. Pour schématiser, on pourrait dire qu'une tâche est un processus soumis à des contraintes temporelles d'exécution.

7.2.1 Les tâches

Le langage Ada propose donc des structures syntaxiques pour décrire des objets ou/et types de tâches. Lorsqu'on définit un type de tâche, les objets tâches devront être déclarés explicitement ou créées dynamiquement. Une tâche comporte deux aspects distincts :
- d'une part, la définition de son interface : il s'agit de définir les points de communication par rendez-vous entre cette tâche et les autres tâches "clientes" sous la forme de déclarations d'entrées. Ces entrées constituent les points de rendez-vous acceptés par la tâche. La syntaxe générale de cette déclaration d'interface est la suivante[6] :

```
task [type] <nom de tâche> is
 entry <nom d'entrée>[( <liste de paramètres> ) ] ;
  ...
end <nom de tâche> ;
```

  À titre d'exemple, l'objet tâche (ou le type de tâche si le mot clé type est présent) calculette définit des entrées d'une tâche acceptant d'exécuter par rendez-vous des opérations arithmétiques :

```
task [type] calculette is
   entry add( a,b in integer ; c out integer);
   entry sub( a,b in integer ; c out integer);
    ...
end calculette;
```

[6] les zones entre crochets sont facultatives

Lorsqu'une tâche cliente veut appeler un rendez-vous sur une tâche serveur, la syntaxe d'appel consiste simplement à préfixer le nom de l'entrée par la référence à la tâche. Par exemple, le rendez-vous add sur la tâche calculette est invoqué par calculette.add(x,y,r).
Si calculette est définie comme un type, alors il faudra déclarer un objet tâche de façon similaire à la déclaration d'une variable quelconque : var uneCalculette : calculette ;, soit même créer dynamiquement une tâche repérée par un pointeur :

```
uneCalcDyn : access calculette;
...
begin
     uneCalcDyn := new calculette ; calculette.add(x,y,r) ;
end
```

– d'autre part, la description du code de la tâche, autrement dit son implantation. En fait, d'un point de vue purement syntaxique, l'implantation d'une tâche se présente sous la forme d'une syntaxe voisine de celle d'une procédure le mot clé procédure étant remplacé par task body. La tâche doit accepter les entrées déclarées dans son interface. Pour ce faire, l'instruction accept permet à une tâche de se mettre en attente d'un appel. La tâche précédente calculette pourrait avoir l'implantation suivante :

```
task body calculette is
   loop
      select
           accept add( a,b in integer ; c out integer) do c := a + b ; end;
      or
           accept sub( a,b in integer ; c out integer) do c := a - b ; end;
        ...
       end select;
  end loop;
end calculette;
```

La structure de contrôle select permet à la tâche d'accepter (de se mettre en attente d'acceptation d') un appel sur une des entrées add, sub,...de la tâche. Ce non déterminisme est très important, l'ordre d'occurrence des appels étant inconnu.

7.2.2 Synchronisation par rendez-vous

La synchronisation par rendez-vous entre tâche appelante et appelée prend plusieurs formes différentes. Nous exposons ces différentes variantes côté émetteur et récepteur.

Appel avec délai de garde Une tâche cliente peut souhaiter renoncer au rendez-vous si celui-ci n'a pas pu avoir lieu dans un délai fixé à l'avance. Pour cela, un appel peut comporter un délai de garde exprimé sous la forme suivante :

```
select
    <référence de tâche>.<entrée>(...);
or delay d ;
end select ;
```

Remarque : On notera la réutilisation malheureuse du mot clé select avec une sémantique différente de celle utilisée pour l'acceptation côté serveur.

Acceptation multiple L'exemple de la tâche calculette montrait un usage de l'instruction select. Sous sa forme générale, l'instruction select a la syntaxe suivante :

```
select
   [ when G_1  => ]  accept <entrée-1>(...) [ do ... end <entrée-1>; ]
                     [ <Bloc-1> ]
    ...
or
   [ when  G_i  => ]  accept <entrée-i>(...) [ do ... end <entrée-i>; ]
                      [ <Bloc-i> ]
   ...
or
   [ when  G_n  => ]  accept <entrée-n>(...) [ do ... end<entrée-n>; ]
                      [ <Bloc-n> ]
end select
```

Sémantique

L'instruction select se déroule de la façon suivante :

1. Chaque garde G_i est évaluée une seule fois déterminant l'ensemble des entrées "acceptables". L'ensemble des entrées acceptables est celles qui ont une garde vraie. Autrement dit, l'instruction select débute par l'exécution du code suivant :

   ```
   acceptables = new Set();
   if (G_1) acceptables.add(entrée-1);
   ...
   if (G_n) acceptables.add(entrée-n);
   if (acceptables.empty()) raise("select erroné");
   ```

   Remarques
   - Lorsqu'une clause when est absente, la garde implicite est when true.
   - Si aucune garde n'est vraie lors de l'évaluation des gardes, alors une exception est levée puisque l'ensemble des entrées acceptables est vide.

2. La primitive d'acceptation multiple du noyau Ada est invoquée en donnant l'ensemble d'entrées précédemment évalué. Si un ou plusieurs appels acceptables est (sont) en attente, un rendez-vous commence en choisissant l'un des appels s'il en existe plusieurs. Les critères de choix ne sont pas fixés par la sémantique du langage.

   Si aucun appel n'est présent sur l'ensemble des entrées acceptables, alors la primitive est bloquante. La tâche appelante est donc momentanément bloquée en attente d'un appel sur l'une des entrées acceptables. Lorsqu'un tel appel arrive, un rendez-vous débute.

3. L'exécution immédiate ou retardée du rendez-vous choisi consiste pour la tâche appelée à exécuter, s'il existe, le bloc do ...end. Sur terminaison de ce bloc, la tâche appelée est prévenue de la terminaison du rendez-vous. La tâche acceptante poursuit alors son exécution par le bloc <Bloc-i> s'il existe. Après quoi, l'instruction select se termine.

Exemple d'une tâche sémaphore On considère une tâche exportant les primitives d'un sémaphore :

```
task Semaphore is
   entry P();
   entry V();
end Semaphore ;
```

Sa réalisation peut être décrite par le code suivant :

```
task body Semaphore is
  cpt : integer := N;
  begin
    loop
      select
        when cpt > 0 => accept P();  cpt := cpt - 1 ;
      or
        accept V(); cpt := cpt + 1 ;
      end select;
    end loop;
end Semaphore ;
```

La tâche de synchronisation initialise la valeur du sémaphore à N et accepte ensuite des rendez-vous sur l'entrée $P()$ si et seulement si la valeur du compteur du sémaphore est positive. Les rendez-vous consistent en fait seulement à réaliser une signalisation entre tâches clientes désirant se synchroniser selon la sémantique d'un objet sémaphore et la tâche de synchronisation.

Acceptation immédiate La structure de contrôle select précédente peut se terminer par une clause else.

```
select
   [ when G_1  => ]  accept <entrée-1>(...) [ do ... end <entrée-1>; ]
                     [ <Bloc-1> ]
    ...
or [ when  G_n  => ]  accept <entrée-n>(...) [ do ... end <entrée-n>; ]
                     [ <Bloc-n> ]
else [ <Bloc par défaut> ]
end select
```

Sémantique

S'il existe un appel acceptable en attente, un rendez-vous a lieu. Mais, s'il n'existe aucun appel acceptable, les instructions de la clause else sont exécutées par défaut (aucun rendez-vous n'étant exécutable immédiatement). Dans ce cas, l'instruction select n'est donc jamais bloquante.

Acceptation avec délai de garde La structure de contrôle select peut se terminer par une clause delay.

```
select
   [ when G_1  => ]  accept <entrée-1>(...) [ do ... end <entrée-1>; ]
                     [ <Bloc-1> ]
    ...
or [ when  G_n  => ]  accept <entrée-n>(...) [ do ... end <entrée-n>; ]
                     [ <Bloc-n> ]
or delay d ; [ <Bloc par défaut> ]
end select
```

Sémantique

Dans ce cas, si aucun rendez-vous ne peut être exécuté immédiatement ou dans le délai maximum fixé, alors le bloc instruction associé à la clause or delay est exécutée par défaut. Aucun rendez-vous n'a pu être exécuté dans le délai imparti.

Acceptation avec clause terminate La structure de contrôle select peut comporter une clause terminate. Cette clause s'insère dans une branche de select de façon naturelle :

```
select
     ...
     or [ when G => ] terminate ;
     ...
end select
```

Sémantique

L'utilisation de cette clause est utile pour assurer la terminaison globale d'une application mettant en œuvre un ensemble de tâches ayant un comportement de type clients-serveurs. Lorsqu'un programme Ada s'exécute, des tâches peuvent être créées dynamiquement. Une tâche est toujours créée dans un bloc d'activation courant constituant son contexte d'exécution[7]. L'imbrication des blocs d'activations engendre un arbre de tâches. Supposons que l'exécution d'une procédure active un ensemble de tâches locales. Alors, cette procédure ne pourra se terminer que lorsque **TOUTES** ces tâches locales seront terminées.

Considérons l'exemple suivant :

```
procedure Application is
   -- déclaration d'un objet sémaphore --
   task Semaphore is entry P(); entry V(); end Semaphore ;
   task body Semaphore is begin loop ...  end loop; end Semaphore ;
   -- déclaration de tâches clientes
   task type cliente;
   Clientes : array(1..4) of cliente ;
   task body cliente is begin  ... ; Semaphore.P(); ...  end cliente ;
begin
  put("Début de la procédure"); new_line;
  put("Fin de la procédure"; new_line;
end Application ;
```

Un appel à la procédure Application ne se terminera que lorsque les tâches locales sémaphore et clientes seront terminées (celles-ci sont créées en prologue de l'exécution du bloc procédural). Or, supposons que les 4 tâches clientes se terminent effectivement. Il existe alors une tâche sémaphore qui ne peut plus recevoir d'appel et qui reste bloquée en attente d'une demande de rendez-vous sur la ou les entrées acceptables du select, ici P et/ou V. La procédure ne se terminera donc jamais.

La clause terminate permet d'éviter ce genre de problème. En effet, si elle existe, l'attente d'un rendez-vous est conditionnée par l'existence de tâches locales actives. S'il n'existe plus que des tâches en attente de rendez-vous sur une instruction select ayant une clause terminate, alors la terminaison de ces tâches est forcée. L'exemple de la tâche sémaphore est un cas particulier où une seule tâche de ce type existe. Si l'on avait créé plusieurs sémaphores, toutes les tâches correspondantes finiraient par être dans ce même état et leur terminaison serait forcée. La terminaison de toutes les tâches locales entraîne finalement la terminaison de la procédure.

À propos des gardes Les expressions booléennes permises dans les gardes ne doivent en principe comporter que des variables locales. Référencer des variables externes (non locales à la tâche) est dangereux dans la mesure où ces variables externes peuvent être modifiées par d'autres tâches. Il ne faut pas oublier que les gardes d'une instruction select ne sont évaluées qu'une fois en prologue de l'exécution de l'instruction. Il est donc préférable que cette condition soit stable lorsque la tâche est en attente de rendez-vous.

[7] Le langage Ada est un langage à structure de blocs avec imbrication possible.

Par ailleurs, nous avons vu que le choix d'un rendez-vous, lorsque plusieurs sont en attente, n'est pas fixé par la sémantique du langage. Il peut donc être intéressant d'imposer certaines priorités entre de tels rendez-vous. Pour cela, on dispose d'un attribut count défini sur les entrées. En fait l'attribut count fournit le nombre d'appels en attente sur l'entrée correspondante. La référence <entrée>'count permet donc de tester s'il existe ou non un rendez-vous possible sur l'entrée correspondante.

Exemple Dans l'exemple suivant, un rendez-vous sur l'entrée $e1$ ne peut pas avoir lieu s'il existe un appel en attente sur l'entrée $e2$. On donne ainsi une priorité d'acceptation aux rendez-vous sur l'entrée $e2$.

```
select
   when (e2'count = 0) => accept e1 ...
or
   accept e2 ...
end select
```

A propos des entrées Il est possible de définir des familles d'entrée. Syntaxiquement, il s'agit de définir un tableau d'entrées :

```
entry <nom d'entrée> (<dimension>) [ (<liste de paramètres>) ] ;
```

À titre d'exemple, un tâche allocateur peut comporter une famille d'entrées associée à l'opération d'allocation, la sémantique de l'indice du tableau étant le nombre de ressources demandées[8]

```
task Allocateur is
               entry Allouer(1..P) ;
               ...
end Allocateur ;
```

La tâche allocateur peut alors se mettre en attente sur une ou plusieurs des entrées de la famille. Néanmoins, Il s'agit bien d'une acceptation sur des indices fixées de la famille. Si l'on veut accepter potentiellement toutes les entrées, il faudra utiliser un select comportant une énumération de toutes les entrées de la famille :

```
select
   accept Allouer(1) do ...
or
   accept Allouer(2) do ...
...
or
   accept Allouer(P) do ...
end select
```

Par contre, il est possible d'accepter une entrée de la famille spécifiée par une variable locale de la tâche :

```
task body Allocateur  is
           ... ;  i := i + 1 ;    accept Allouer(i) do ...
end Allocateur ;
```

En conclusion, la notion de famille d'entrées est utile lorsque l'on veut distinguer une priorité entre tâches appelantes. Ce mécanisme complète les possibilités offertes, en la matière, par l'attribut count utilisé dans les gardes d'un select.

[8] Attention : le domaine d'indice $1..P$ n'est pas un paramètre formel. La syntaxe est ambiguë en apparence seulement. Le champ entre parenthèses est une déclaration de type discret et non pas une liste de paramètres formels.

7.2.3 Tâches et automates

Une tâche de synchronisation peut être interprétée comme un automate d'états fini. Elle possède un état et selon son état, elle accepte un certain nombre de transitions représentées par les entrées acceptables. Cette interprétation permet de programmer des tâches de façon assez systématique dans la mesure où il suffit de traduire l'automate en instructions Ada. L'automate peut être décrit par une boucle dans laquelle une instruction case permet de fixer la ou les transitions possibles (acceptables) à partir de l'état courant. Le schéma général est le suivant pour un automate déterministe :

```
task body automate is
  begin
     loop
       case etat is
         ...
         -- transition unique --
         when etat = s => accept e ; etat := ... ;
         -- transitions multiples --
         when etat = s' =>
                select
                     accept e' ; etat := ... ;
                or
                     accept e" ; etat := ... ;
                or
                     terminate
                end select
         ...
       end case
     end loop
  end automate ;
```

case var is when val1 =¿ ... when val2 =¿ ... end case ;

8 Approche par aspects

Les problèmes de synchronisation ont donné naissance à différents mécanismes. Les langages de programmation à objets ont, quant à eux, mis en avant la réutilisation, la modularité et l'encapsulation. Une idée intéressante est alors d'essayer de séparer la description des aspects purement fonctionnels d'un objet partagé d'une part, et d'autre part, la description de la synchronisation des opérations sur l'objet. La programmation par aspects adopte une telle démarche.

Nous illustrons cette approche sur un exemple, celui du producteur-consommateur. Un langage à objet tel que Java permet de décrire un tampon partagé à N cases sous la forme d'une classe.

```
class Tampon {
   Message blocs[] ;
   int taille ; int t = 0 ; int q = 0 ;
   void Déposer(Message Mes) {
      blocs[q] = Mes ; q = (q + 1) % taille ;
   }
   Message Retirer() {
     Message me = blocs[t] ; t = (t + 1) % taille ; return me ;
   }
   Buffer(int N) { taille = N ; blocs = new Message[taille] ; }
}
```

Cette description ne comporte strictement que l'aspect fonctionnel des opérations de dépot et retrait d'un message. On remarquera que les tests d'existence de cases libres ou pleines ne sont pas programmés.

Une deuxième description va préciser les contraintes d'ordonnancement des opérations si un tel objet tampon est partagé. Cette « capsule » appelée coordinateur comporte les définitions suivantes :

- Clause selfex : Garantie de l'exécution d'une opération en tant que section critique. Une clause complémentaire mutex permet de garantir l'exécution en exclusion mutuelle entre des opérations distinctes.[9] ;
- Déclaration des compteurs du nombre d'opérations terminées de chaque type ;
- Clause guard : Bloc précisant pour un opérateur les conditions d'exécutabilité (clause require) et comportant éventuellement un code prologue (clause onentry) et épilogue (clause onexit) ;

```
coordinator of Tampon {
   invariant 0 <= #Deposer - #Retirer <= taille
   mutex Déposer, Retirer ;
   selfex Déposer, Retirer ;
   int #Deposer = 0 ;  #Retirer = 0 ;
   guard Déposer :
     require #Déposer - #Retirer < taille ;
     onexit { #Déposer++ ; }
   guard Retirer :
     require #Déposer - #Retirer > 0 ;
     onexit { #Retirer++ ; }
}
```

Un logiciel "tisseur" (weaver) aura la charge de recoller les morceaux pour obtenir un objet partageable correctement synchronisé.

9 Systèmes parallèles à ressources critiques et interblocage

Les systèmes d'exploitation ont constitué les premiers systèmes parallèles à ressources critiques. En effet, les processus exécutant des programmes utilisateurs en parallèle entrent en conflit pour utiliser les ressources critiques telles que les périphériques (imprimantes, traceurs, dérouleurs de bandes, lecteurs/enregistreurs de DVD, etc) mais aussi pour obtenir de l'espace mémoire ou l'accès à un fichier.

9.1 Modélisation du problème

Si l'on ne s'intéresse qu'à ce problème de conflit, un système d'exploitation entre donc dans la classe des systèmes parallèles à ressources critiques. Ce genre de système peut être modélisé sous la forme d'un ensemble de processus et de classes de ressources comme dans la figure (2).

Un tel système peut alors être abstrait sous la forme d'un graphe. Les sommets peuvent être soit des processus, soit des ressources critiques. Un arc existe seulement entre un processus et une ressource dans deux situations selon son orientation :

- un arc orienté du processus vers la ressource existe lorsque le processus est bloqué sur la demande de la ressource (qui est occupée par un autre processus) ;
- un arc orienté de la ressource vers le processus existe lorsque le processus a obtenu la ressource.

Ce graphe représente l'état du système en ce qui concerne l'allocation des ressources aux processus. La figure (3) montre l'exemple d'un tel graphe captant l'état du système composé de deux processus et deux ressources.

[9] Attention, l'utilisation de la clause mutex seule n'interdit pas l'exécution parallèle de plusieurs opérations Déposer par exemple.

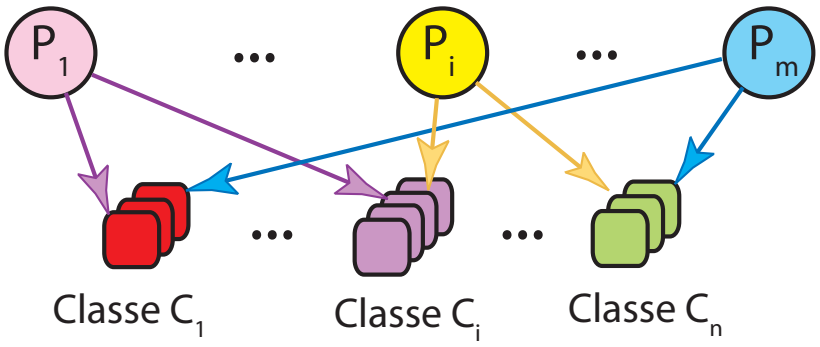

FIG. 2 – Systèmes parallèles à ressources critiques

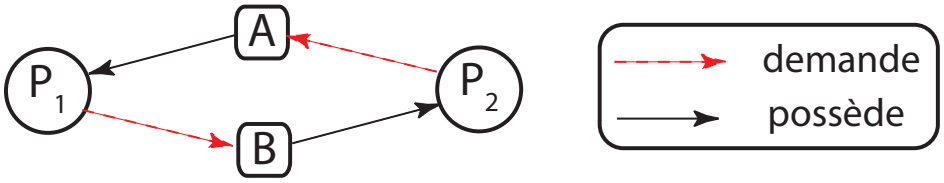

FIG. 3 – Interblocage entre deux processus

Le code des deux processus est de la forme suivante :

```
process P1() {                    process P2() {
          ...                               ...
          demander(A) ;                     demander(B) ;
  dem1 :  demander(B) ;             dem2 :  demander(A) ;
          ...                               ...
          restituer(B) ;                    restituer(A) ;
          restituer(A) ;                    restituer(B) ;
}                                 }
```

Le graphe illustre une situation possible atteinte par ce système lorsque les processus en sont arrivés aux points de contrôle dem1 pour P_1 et dem2 pour P_2. Le processus P_1 possède la ressource A mais est bloqué sur la demande de la ressource B. En effet, cette ressource B est attribuée au processus P_2 qui a lui même à son tour demandé la ressource A. On constate qu'il existe alors un cycle dans le graphe d'état courant du système. Ce cycle est moyen simple de détection d'un interblocage.

Remarque Il faut bien noter que l'interblocage n'est qu'un risque. Si le processus P_1 avait été plus rapide et avait obtenu les deux ressources A et B ($\equiv$ s'il avait atteint le point de contrôle dem1) avant que le processus P_2 ait demandé la ressource B, l'interblocage n'aurait pas eu lieu.

Si l'on veut qu'un système n'atteigne pas une situation d'interblocage, il faut donc empêcher la formation d'un tel cycle, les sommets processus appartenant au cycle étant en situation d'interblocage[10].

9.2 Conditions nécessaires à un interblocage

De façon plus rigoureuse, dans un système d'exploitation, un ensemble de processus risquent d'atteindre un état d'interblocage si les quatre conditions (nécessaires) suivantes sont vérifiées :

[10] si rien n'est prévu pour sortir de cet état

- **Criticité** : les processus accèdent à des ressource critiques ;
- **Pas d'abandon** : un processus bloqué sur une demande maintient sa demande et garde les ressources qu'il possède déjà.
- **Pas de préemption** : un processus bloqué sur une demande ne perd pas les ressources qu'il possède. Autrement dit, le système d'allocation de ressources ne préempte pas des ressources possédées par un processus bloqué pour les donner à un autre demandeur ;
- **Cycle** : un cycle est présent dans le graphe d'allocation.

À partir de ces constatations, de nombreuses études ont été faites dans le domaine des systèmes d'exploitation pour traiter le problème. Deux grandes approches sont possibles :
- soit prévenir tout risque d'interblocage. On parle alors naturellement de techniques de prévention.
- soit détecter l'occurrence d'un état d'interblocage. Il faudra alors un algorithme de récupération de la faute pour que le système évolue vers un nouvel état correct.

9.3 Techniques de prévention

Toutes sont fondées sur l'idée qu'il suffit de rendre une des conditions nécessaires fausse en tant qu'invariant durant toute exécution du système. Nous n'exposons pas toutes les techniques mais montrons quelques unes des principales solutions.

9.3.1 Prévention des cycles

La condition la plus simple à envisager est d'éviter la formation d'un cycle dans le graphe. Pour cela, deux approches sont possibles : soit en ordonnant les ressources (classes de), soit en ordonnant les processus.

Stratégie fondée sur l'ordonnancement des ressources La première solution consiste donc à ordonner les ressources par classes [Hav68]. Cet ordre interviendra dans la décision d'accorder ou non une ressource au processus demandeur. La stratégie d'allocation repose sur les principes suivants :
- Toute ressource r appartient à une classe C ;
- Les classes sont totalement ordonnées par une relation notée $\prec$: $C_1 \prec C_2 \prec \ldots \prec C_n$
- Un processus ne peut acquérir des ressources qu'en respectant l'ordre établi entre les classes de ces ressources. Autrement dit, soit C_{Max} la classe maximale parmi les ressources possédées par un processus. Alors, celui-ci ne peut plus demander que des ressources de classe C' telles que $C_{Max} \prec C'$.

Un cycle ne peut alors se former si cette stratégie de prévention est adoptée. En reprenant l'exemple précédemment présenté, si l'ordre choisi sur les ressources est l'ordre lexical, on peut constater qu'une telle stratégie empêcherait le processus P_2 de demander la ressource A après avoir obtenu B évitant ainsi la formation du cycle.

Cette approche a été utilisée dans les systèmes de traitement par lots. Il faut prendre soin de définir, entre les classes de ressources, un ordre qui soit adapté au comportement des processus. Sinon, l'allocation des ressources ne sera plus optimale. Un processus peut être amené à demander une ressource qu'il n'utilisera pas avant un long délai simplement parce qu'il a besoin immédiatement d'une ressource de classe plus élevée.

Stratégie fondée sur l'ordonnancement des processus Dans cette stratégie, un ordre total est attribué aux processus. Ceci peut être fait en affectant une estampille à chaque processus lors de sa création[11] La stratégie d'allocation repose sur les principes suivants :
- Les processus sont totalement ordonnés par une relation notée $\prec$ sur leurs estampilles. Cet ordre peut être interprété comme une priorité pour obtenir les ressources. On note P.e l'estampille du processus P.

[11] Au plus simple, on peut utiliser le numéro attribué à chaque processus par le noyau du système d'exploitation.

– Une ressource possède un attribut état et un attribut prop qui précise l'estampille du processus possédant la ressource lorsque celle-ci est occupée.
– Lors de la demande d'une ressource r par un processus P, on applique l'algorithme suivant :

```
if  (r.état==libre) {Allouer r à P ; r.état=occupée; r.prop=P.e }
elsif (r.prop ≺ P.e) {Maintien de la demande : P est bloqué }
else { La demande de P est rejetée ; }
```

On voit donc que la demande échoue si la ressource est déjà possédée par un processus plus prioritaire. Le processus devra donc renouveler sa demande ultérieurement. Par contre, la demande d'un processus plus prioritaire que le processus actuellement propriétaire de la ressource est mise en attente mais maintenue.

Remarque On peut se poser la question d'un risque de famine pour certains processus. Cependant, un processus devient de plus en plus prioritaire au fur et à mesure que le temps passe et que des processus plus prioritaires sont terminés.

Cette stratégie est surtout appliquée dans les systèmes transactionnels associés aux bases de données. Les estampilles sont associées aux transactions[12] et les ressources sont les bases de données (ou enregistrements).

9.3.2 Prévention par abandon

Cette stratégie consiste à éviter qu'un processus reste bloqué alors qu'il possède des ressources. Pour ce faire, on oblige tous les processus à acquérir leurs ressources par paquets. Par exemple, un processus demande ses ressources lors de sa création. Il doit pouvoir s'exécuter sans demander d'autres ressources au cours de son exécution. Un processus a donc tout ou rien. Cette approche a été utilisée dans les systèmes de traitement par lots en découpant les traitements en étapes selon les ressources nécessaires à l'exécution de cette étape. Bien que simple, cette approche reste donc peu optimale.

9.3.3 Prévention par préemption

Cette stratégie ne peut être appliquée qu'avec certains types de ressources. L'idée de base est qu'un processus bloqué sur sa dernière demande de ressource n'utilise donc pas les ressources qu'il possède. On a donc l'opportunité de lui préempter momentanément ces ressources devenues inutilisées pour les donner à d'autres processus. C'est le noyau de gestion de la classe de ressources considérée qui doit alors prendre en charge cette préemption.

Il existe un type de ressources pour lequel cette stratégie est utilisée. Il s'agit des mémoires virtuelles avec "pages à la demande". Nous n'entrons pas dans les détails de ce genre de système, mais l'idée de base est de préempter des pages de mémoire réelles obtenues par des processus bloqués (sur d'autres conditions d'allocation) pour satisfaire une demande alors qu'il n'y a plus de pages réelles libres.

9.3.4 Prévention par non criticité

Il peut être difficile d'imaginer qu'une ressource critique puisse devenir non critique. En fait une solution existe en substituant à la ressource critique une autre ressource. Cette approche est utilisée en particulier au niveau de l'impression. Une imprimante est naturellement une ressource critique. Une solution adoptée très tôt dans les systèmes d'exploitation a été de remplacer l'imprimante par un fichier. Lorsque le processus exécute des instructions d'impression celles-ci sont redirigées automatiquement vers un fichier au prix d'une désynchronisation des sorties. Une fois le processus terminé, ce fichier est mis en file d'attente d'impression. Un processus unique accède à l'imprimante réelle en prenant les fichiers à imprimer un par un dans la file d'attente. Ainsi, l'imprimante n'est plus partagée par plusieurs processus. Ce genre de système dit

[12] Une transaction est un traitement qui accède à des données atomiquement : aucun état intermédiaire des données modifiées par la transaction ne doit être visible aux autres transactions. Un processus peut exécuter plusieurs transactions successives.

de spooling (de l'abréviation SPOOL : Simultaneous Peripheral Operations OnLine) exploite à la fois une désynchronisation des sorties et une ressource "virtuelle" d'impression représentée par un fichier de données.

9.4 Détection

Les méthodes de détection reposent sur la gestion du graphe d'allocation précédemment décrit. Un graphe simplifié peut parfois être utilisé, celui-ci ne comportant plus que les sommets processus. On appelle un tel graphe, graphe d'attente (Wait-for graph). Dans un tel graphe, les sommets sont donc tous des processus et il existe un arc d'un processus P vers un processus P' ssi le processus P a demandé une ressource possédée par P'. On voit que cette solution implique :

- soit que les demandes soient toujours parfaitement identifiées : le processus demande une ressource précise unique ;
- soit qu'un arc soit ajouté au graphe vers chaque processus possédant une ressource susceptible de satisfaire la demande. Lors de l'allocation d'une des ressources possibles, il faudra alors supprimer tous les arcs inutiles associés à la demande satisfaite.

Quel que soit le graphe géré, la présence d'un interblocage est détectée par un cycle dans le graphe. Une fois la présence d'un cycle détectée, il faut évidemment corriger cet état. Les processus présents dans le cycle ne peuvent plus s'exécuter. Une solution consiste à détruire un des processus ($\equiv$ casser le cycle). Cette solution est cependant rude et peut ne pas satisfaire l'usager concerné.

9.5 Conclusion

Le phénomène d'interblocage a été très tôt découvert dans les systèmes d'exploitation multi-programmés. Bien que le problème soit modélisable simplement à l'aide d'un graphe, les solutions proposées sont multiples. En fait, des solutions particulières ont été élaborées et adaptées pour des grandes classes de ressources (ressources d'impression, bases de données, traitements par lots, mémoires virtuelles, etc). Les systèmes répartis ont amené une complexité de mise en œuvre supplémentaire dans la mesure où le graphe d'allocation pose un problème de construction d'état global.

Références

[BA82] M. Ben-Ari. Principles of concurrent programming. pages 38–43, 1982.

[Bar00] J. Barnes. *Programmer en Ada 95 (2ième Edition).* Langages/Programmation. Vuibert Informatique, 2000.

[BH72] P. Brinch Hansen. Structured multiprogramming. *ACM Communications*, 15(7) :574–578, 1972.

[BH75] P. Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, 1(2) :199–207, 1975.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programmin control. *ACM Communications*, 8(9) :569, Sept. 1965.

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

[Hav68] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2) :74–84, 1968.

[Hoa74] C. A. R. Hoare. Monitors : an operating system structuring concept. *ACM Communications*, 17(10) :549–557, 1974.

[Lam74] L. Lamport. A new solution of dijkstra's concurrent programming problem. *ACM Communications*, 17(6) :453–455, 1974.

[Pad90] G. Padiou, A. Sayah. *Techniques de synchronisation pour les applications parallèles.* Editions CEPADUES, 1990.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3) :115–116, June 1981.