CHAPITRE 8

CONTRÔLE DE CONCURRENCE EN MÉMOIRE COMMUNE
MÉCANISMES PRIMITIFS D'EXCLUSION MUTUELLE

Plan

Manuel 8

Contrôle de concurrence en mémoire commune Mécanismes élémentaires d'exclusion mutuelle

1. Contrôle de concurrence en mémoire commune

1.1. Architecture multiprocesseur et systèmes centralisés

La concurrence de processus peut se produire dans deux classes d'architecture de systèmes : systèmes centralisés ou systèmes répartis. Nous étudions ici le cas des systèmes centralisés, c'est à dire ceux qui contrôlent la concurrence des processus et le partage des ressources en utilisant des données communes en accès direct. L'architecture matérielle qui supporte ce type de systèmes est une architecture centralisée avec une mémoire centrale commune partagée par plusieurs processeurs, appelée architecture multiprocesseur. Un cas particulier fréquent est celui du monoprocesseur.

On suppose, pour simplifier l'étude et rester au niveau logique, et ne pas avoir à se préoccuper de l'allocation des processeurs aux processus, qu'il y a autant de processeurs que de processus. Cependant quand l'influence de l'allocation pourra être utile pour faire comprendre certains phénomènes qui résultent de l'inactivité d'un processus prêt, nous nous placerons alors dans le cas monoprocesseur.

On verra plus loin le cas des systèmes répartis communiquant uniquement par messages entre processus concurrents.

1.2. Accès atomique à une cellule de mémoire centrale partagée

Dans ce contexte, le phénomène physique élémentaire qui est utilisé est l'accès d'une instruction machine qui fait référence à une cellule de cette mémoire commune. Cet accès, qu'il soit en lecture ou en écriture est construit de telle façon qu'il n'y ait qu'un seul processeur à la fois qui puisse le faire. L'opération d'accès est atomique. Si plusieurs processeurs lancent en même temps un ordre d'accès sur le bus qui conduit à la mémoire commune, un seul accès est pris en compte, et les autres sont mis en attente pour être traités séquentiellement. Cela entraîne le phénomène bien connu de congestion du bus mémoire d'un multiprocesseur. On lutte contre cette congestion en construisant une architecture qui comprend plusieurs modules de mémoire partagée, chaque module ayant son bus d'accès.

En fait le phénomène physique élémentaire fondamental est la mise en mémoire sous forme binaire. Comme la mémoire est binaire, le résultat d'une écriture doit être 0 ou 1, que l'écriture soit concurrente ou non. Dans ce cas toute lecture donne aussi 0 ou 1. On pourrait utiliser ce phénomène (obtenu par le réglage des seuils de valeurs des tensions électriques déterminant les valeurs 0 et 1) pour construire les mécanismes élémentaires que l'on va étudier.

On utilise cependant toujours les accès élémentaires câblés et associés aux instructions machine élémentaires d'écriture ou de lecture à une adresse de la mémoire commune. Dans une architecture multiprocesseur avec un cache pour chaque processeur, les caches sont mis à jour à chaque modification pour avoir les mêmes valeurs pour chaque copie.

On va étudier, dans ce contexte, les solutions au paradigme de l'exclusion mutuelle.

Une première série de solutions entraînera l'attente active des processus qui se détectent en conflit d'exclusion mutuelle :

- On verra d'abord à n'utiliser que l'écriture et la lecture atomiques.
- On utilisera ensuite une instruction spéciale apportant une utilisation atomique du bus mémoire pour deux accès consécutifs, l'un en lecture, l'autre en écriture (c'est le "test and set").
- On verra aussi l'influence des interruptions machine et leur nécessaire masquage.

On construira alors un mécanisme plus complexe qui lorsqu'un processus est en conflit d'exclusion mutuelle, bloque ce processus et lui réquisitionne le processeur, à charge de le réveiller ultérieurement, à bon escient, et de lui permettre de recevoir à nouveau le processeur. On étudiera plus particulièrement le mécanisme appelé sémaphore. On donnera également un aperçu des moniteurs en java et des objets protégés en Ada 95.

On situera les solutions dans le contexte de la bonne programmation, c'est à dire en considérant une programmation modulaire.

1.2. Rappel du paradigme de l'exclusion mutuelle

Lorsque plusieurs processus font accès à une ressource critique en utilisant chacun une séquence d'instruction, appelée section critique du processus, cet accès doit être contrôlé de telle sorte que lorsque les demandes d'accès sont concurrentes, un processus au plus puisse exécuter sa section critique.

Toute solution suppose un protocole d'initialisation. Chaque processus appelle une procédure de contrôle d'entrée en section critique, puis une procédure de sortie de section critique. La procédure d'entrée doit permettre à un processus d'entrer en section critique, mais à un au plus, et les autres processus doivent attendre. La procédure de sortie doit permettre de signaler aux processus en attente qu'il n'y a plus de processus en section critique, et éventuellement laisser entrer un et un seul processus dans sa section critique.

Les solutions ne doivent dépendre ni des vitesses, ni des priorités ni de l'ordonnancement des processus. La ressource critique doit pouvoir être utilisée si plusieurs processus la demandent (pas d'interblocage entre demandeurs). On souhaite parfois que les processus puissent utiliser la ressource de façon équitable (pas de famine d'un demandeur).

Par hypothèse, tout processus sort de section critique au bout d'un temps fini. C'est une hypothèse de fiabilité minimale qui devra être certifiée pour la programmation des processus du noyau du système, ou qui sera assurée par ce noyau pour les processus des utilisateurs.

2. Attente active

2.1. En l'absence de mécanisme

Un retour sur l'exemple du compte client permet de rappeler que la valeur finale de la variable commune peut varier de 2 à 32 selon l'ordonnancement des processus.

En notant [ ]^x une boucle de x cycles, on peut tracer quelques ordonnancements significatifs. Il vient :

{COMPTE_CLIENT = 0} [P1 P2 P3]^{16} [Q1 Q2 Q3]^{16} {COMPTE_CLIENT = 32}

{COMPTE_CLIENT = 0} [P1 P2 P3 Q1 Q2 Q3]^{16} {COMPTE_CLIENT = 32}

{COMPTE_CLIENT = 0} [P1 Q1 P2 P3 Q2 Q3]^{16} {COMPTE_CLIENT = 16}

{COMPTE_CLIENT = 0}P1[Q1 Q2 Q3]^{15} P2 P3 Q1[P1 P2 P3]^{15} Q2 Q3{COMPTE_CLIENT = 2}

2.2. Mécanisme trop naïf pour être correct

Une première idée est, puisqu'on dispose de variables communes, d'en utiliser une pour noter l'occupation de la ressource commune. Soit Verrou cette variable booléenne. Quand Verrou est Vrai, la ressource est déjà occupée et le processus doit attendre par une boucle d'attente jusqu'à ce que Verrou prenne la valeur Faux. Si Verrou est Faux, le processus prend le Verrou en le notant à Vrai et entre en section critique.

L'ennui c'est que deux processus peuvent lire le Verrou à Faux avant que l'un d'eux l'ait mis à Vrai. Et ces deux processus entrent de concert en section critique, ne respectant pas l'exclusion mutuelle. Cela provient de ce que la lecture comme l'écriture de Verrou sont des opérations atomiques, mais que la séquence, lecture suivie d'écriture, n'est pas atomique. On verra plus loin avec le "test and set" qu'on a introduit une telle opération câblée.

Donc avec seulement les opérations atomiques d'écriture ou de lecture, une seule variable commune ne suffit pas. Il faut trois variables.

2.3. Démarche pour une solution correcte avec deux processus

Chacun des deux processus fait connaître sa candidature et avant d'entrer en section critique vérifie que l'autre n'est pas demandeur. Si c'est le cas il se place dans une boucle d'attente. Le danger est que les deux processus soient candidats en même temps et s'attendent mutuellement. c'est un interblocage.

Il faut casser cet interblocage. On peut imaginer que chaque processus retire sa candidature un temps puis la renouvelle. Rien ne garantit qu'on évitera toujours le cas extrême où les deux processus bouclent en synchronisme, retirant et renouvelant en même temps leurs candidatures.

On peut s'en approcher, comme il est fait pour gérer les collisions dans les réseaux locaux de type Ethernet, en programmant le retrait de chaque processus pour une durée distincte, obtenue par une boucle de comptage. La valeur à compter pendant l'attente est soit fixe et différente pour chaque processus, soit tirée au hasard. Cependant rien ne garantit que l'ordonnancement des processus ne perturbe la durée prise pour le comptage. On est en temps chronologique dans le processus alors qu'il faut un temps chronométrique indépendant des processus. Ce "temps réel" ne peut être obtenu que par une horloge externe, une par processus. S'il n'y a qu'une horloge commune, c'est une ressource partagée à utiliser en exclusion mutuelle, problème non résolu!! Autre ennui avec cette solution, elle n'est pas équitable quand chaque processus reçoit une durée d'attente différente. Celui qui reçoit la plus petite durée devient prioritaire.

En cas de candidatures simultanées, on règle le conflit en donnant la priorité à un des deux processus, mais, pour que la solution soit équitable et se rapproche d'un service à l'ancienneté des candidatures, la priorité doit changer d'une candidature à la suivante.

Ce souci d'équité rend le problème très difficile. C'est un excellent exemple des difficultés que l'on rencontre quand on doit contrôler des processus concurrents. C'est pourquoi on le traite en détail.

2.4. Algorithme de Dekker (la plus ancienne solution équitable, deux processus)

En cas de candidatures simultanées, on utilise une variable Premier qui indique quel est le processus qui doit passer en premier. Sa valeur doit rester fixe tant que ce processus n'est pas sorti de section critique. Le processus non prioritaire doit retirer sa candidature. Le processus prioritaire attend ce retrait avant d'entrer en section critique. Le processus non prioritaire ne doit pas renouveler sa candidature avant que la variable Premier ait été modifiée par l'autre processus qui lui indique ainsi que c'est son tour. C'est une solution difficile à valider car il faut respecter l'exclusion mutuelle mais aussi empêcher qu'un processus très rapide ne redevienne candidat et s'interclasse avec les opérations d'un processus lent à entrer en section critique. On doit, pour assurer la preuve, obliger les deux processus à passer toujours par un état de référence où il n'y a qu'un seul candidat.

Cet algorithme date de 1965 et est la première solution publiée. Elle se généralise difficilement au cas de plusieurs processus (N connu quelconque)

2.5. Algorithme de Peterson (la meilleure solution pour deux processus)

En cas de candidatures simultanées, on utilise une variable Dernier qui indique quel est le processus qui doit passer en dernier. Sa valeur est changée dynamiquement par le dernier processus qui se déclare candidat, immédiatement après cette déclaration. En cas de candidatures simultanées, on bloque le dernier demandeur jusqu'à l'arrivée d'un nouveau dernier ou jusqu'au retrait de la candidature de l'autre. Cette solution est subtile car tout repose sur cet ordre, sur le fait que tout nouveau candidat se propose comme victime et que cette situation ne peut durer indéfiniment. Tout processus qui sort de section critique et redevient candidat devient dernier et doit attendre que l'autre soit passé avant lui. C'est donc une solution équitable. La preuve de cette solution est beaucoup plus simple.

Cet algorithme date de 1981. C'est une solution optimale, en ce sens que c'est la solution qui se modélise avec le plus petit graphe d'états. Elle se généralise bien pour N processus. L'algorithme doit considérer N-1 étapes successives et à chaque étape il faut éliminer un candidat. À la fin, au bout de N-1 étapes au plus, il ne reste qu'un candidat.

2.6. Algorithme de Lamport (solution pour N processus)

L'intérêt de cet algorithme est qu'il en existe une version adaptée à la répartition et à la communication par messages (on la verra au chapitre 10) ;

1. Chaque processus candidat commence par enregistrer sa candidature en prenant une estampille (un Ticket). Un processus non candidat a une estampille nulle. La règle est de prendre l'estampille qui suit la plus grande estampille déjà attribuée, pour créer une file de candidature avec une attente à l'ancienneté. Le processus peut consulter pour cela toutes les estampilles attribuées aux autres candidats. Considérons les effets de la concurrence :

a) Deux processus très bien synchronisés peuvent lire tous deux les mêmes valeurs, trouver 0 comme valeur du concurrent, calculer donc la même valeur maximale. Ils auront ainsi la même estampille. On verra plus loin comment les départager.

b) Supposons qu'une estampille est remise à zéro par le processus qui vient de sortir de section critique. Un processus i peut ainsi prendre une estampille qui sera plus grande que celle que prendrait un concurrent j arrivant un peu après lui pour prendre une estampille. Cela le retardera un peu, mais ce ne sera pas plus grave.

c) Un processus qui prend une estampille peut aussi calculer une estampille plus petite qu'un concurrent arrivé avant lui. Mais une fois le choix fait, les valeurs des estampilles ne changent plus et restent stables. Il faut savoir repérer et attendre la fin de cette prise d'estampille. On utilise pour cela une variable booléenne Choix, une par processus, qui indique que le processus acquiert une estampille. Pendant cette phase d'acquisition, l'estampille du processus est inutilisable pour classer ce processus parmi les autres (voir phase suivante).

Définition : On dit qu'un processus est en phase d'enregistrement tant que sa variable Choix est à Vrai.

2. Chaque processus candidat se place dans l'ordre d'attente en comparant son estampille à celle de tous les autres. C'est la phase d'ordonnancement. Il n'y a pas de file d'attente

centralisée, car il faudrait une exclusion mutuelle pour la gérer, mais chaque processus attend quand il repère un processus qui a une estampille plus petite que la sienne. Au bout de la consultation de tous les autres processus, chaque processus est sûr d'avoir la plus petite estampille de tous, car tous ceux qui avaient une estampille plus petite que lui sont passés en section critique et ont remis à zéro leur estampille ; il a acquis le droit d'entrer en section critique. L'ancienneté des candidatures est déterminée par la valeur des estampilles des processus enregistrés. Tout processus qui se compare à un processus en phase d'enregistrement doit attendre la fin de cet enregistrement. On montre que l'algorithme est correct, quels que soient les effets de la concurrence :

a) Si deux processus ont la même estampille, on les départage en prenant comme critère une priorité fixe qui peut être l'identificateur unique du processus. Il est nécessaire que cette priorité donne un ordre total aux processus (on le note =>), donc que deux processus n'aient pas la même priorité.

b) Les estampilles ne sont pas des entiers consécutifs. On les compare deux à deux.

c) Un processus ne risque pas de passer avant un candidat qui a une estampille plus petite que la sienne (obtenue après la comparaison des valeurs).

Preuve de cette assertion.

Définition : On dit qu'un processus est en phase d'ordonnancement dès qu'il a fini son enregistrement et jusqu'à ce qu'il sorte de section critique.

T1 : Si deux processus i et k sont en phase d'ordonnancement et que i est entré dans cette phase avant que k ne soit entré en phase d'enregistrement, alors l'estampille de i est plus petite que celle de k.

En effet l'estampille de i était enregistrée avant que k ne calcule la sienne. Il n'y avait pas concurrence entre i et k pendant l'enregistrement.

T2 : Si le processus i est en section critique et si le processus k est en phase d'ordonnancement, alors l'estampille de i est plus petite que celle de k ou bien elles sont égales et la priorité de i est plus petite que la priorité de k.

En effet si le processus i a constaté que Choix[k] était faux, cela a pu signifier deux choses et deux seulement :

a) k venait de finir de calculer son estampille, et celle-ci était nécessairement supérieure à celle de i puisque i a pu continuer ses comparaisons avec les processus suivants

b) k ne s'était pas encore enregistré et alors, d'après T1 ci-dessus, son estampille sera plus grande que celle de i s'il décide de s'enregistrer avant que i ne sorte de section critique.

2.7. Instruction spéciale atomique (ou indivisible)

Pour simplifier et obtenir une solution qui rende correcte l'approche vue au 2.2., des processeurs fournissent des instructions câblées spéciales dont le rôle est de rendre atomique la séquence : lecture suivie d'écriture. Il devient alors impossible à deux processeurs exécutant des processus concurrents de lire le Verrou à Faux avant que l'un d'eux l'ait mis à Vrai. Les instructions câblées atomiques (on dit aussi indivisibles) les plus usitées sont TAS ("test and set") et SWAP ("exchange to memory"), que l'on peut décrire comme suit :

```
function TAS (M : in out Boolean) return Boolean is
-- instruction câblée indivisible
 Copie : Boolean; -- variable interne à l'instruction dans l'unité centrale qui l'exécute
begin Copie := M; M := True; return Copie; end TAS;
```

Pendant l'exécution de l'instruction, le bus d'accès à la mémoire partagée qui contient M est réservé à un et un seul processeur pendant ces 2 accès : indivisibilité de l'opération.

```
procedure SWAP (A, B : in out Boolean) is
-- instruction câblée indivisible
 Copie : Boolean; -- variable interne à l'instruction dans l'unité centrale qui l'exécute
begin Copie := A; A := B; B := Copie; end SWAP ;
```

L'algorithme a l'avantage d'être valable pour un nombre quelconque de processus, mais ce n'est pas un algorithme équitable puisqu'un processus très rapide qui vient de sortir de section critique peut y entrer de nouveau avant tous les autres. Un algorithme équitable est indiqué en exercice.

2.8. Masquage des interruptions en monoprocesseur

Dans une architecture monoprocesseur, les processus partagent le processeur sous le contrôle de l'unité centrale et réalisent une pseudo-concurrence. Un processus peut perdre l'utilisation du processeur à la suite d'une fin de quantum (déclenchée par l'interruption d'horloge), de l'activation d'un processus plus prioritaire, par le déclenchement d'un sous-programme d'interruption. Pour garder l'exclusion mutuelle entre les sections critiques, il suffit d'interdire la commutation de processus pendant qu'un processus est en section critique, ou encore de masquer les interruptions (pour le cas où un sous-programme d'interruption aurait une ressource critique partagée avec un processus. Ce cas est fréquent car souvent un sous-programme d'interruption capture des données communes exploitées ensuite par un processus réveillé pour le faire. Si on ne masque pas les interruptions, on aboutit à un interblocage entre d'une part le processus qui a la ressource critique, mais n'a plus le processeur et d'autre part le sous-programme d'interruption qui garde le processeur et qui attend la fin de la section critique du processus).

Toutefois, le masquage et démasquage des interruptions ne peut être fait qu'en mode maître, donc dans le noyau. Il faut prévoir un appel système spécifique.

D'autre part les sections critiques sont souvent longues et pendant leur exécution plusieurs interruptions successives peuvent être déclenchées sur le même niveau d'interruption, et elles ne sont mémorisées que comme une interruption et les autres sont perdues. Cela peut en particulier retarder l'horloge du système. D'autre part si on a des entrées-sorties pendant une section critique, on a besoin des interruptions pour traiter ces entrées-sorties.

Le masquage et démasquage des interruptions sont donc utilisés pour des sections critiques très courtes dans le noyau, et en particulier pour celles qui servent à coder des mécanismes de synchronisation pour les programmes utilisateurs, comme les sémaphores, ou la synchronisation de base des langages comme Ada ou Java.

2.9. Instruction spéciale atomique et masquage des interruptions

Dans une architecture multiprocesseur, le masquage des interruptions protège seulement des événements externes. Il faut aussi contrôler la concurrence interne entre processeurs. On associe le masquage d'interruptions et un algorithme d'exclusion mutuelle avec attente active, par exemple celui avec le "test and set". On a une suite de deux actions et leur ordre n'est pas indifférent. Il faut masquer les interruptions en premier avant la demande d'entrée en section critique, sinon une interruption pourrait réquisitionner le processeur d'un processus autorisé à entrer en section critique mais non encore masqué contre les interruptions

2.10. Bilan de l'attente active : inconvénients et utilisation

L'attente active a comme inconvénient d'immobiliser le processeur simplement pour attendre.

En monoprocesseur, on risque d'attendre indéfiniment. Il faut que le processus en attente libère le processeur explicitement (primitive système "sleep()" en Unix, "yield()" en Java, "delay" en Ada);

En multiprocesseur, non seulement on gâche de la puissance disponible, mais on congestionne le bus mémoire avec des accès pour vérifier que l'attente est toujours nécessaire et cela ralentit les accès mémoire, donc a puissance de calcul, du processeur en section critique.

L'attente active est cependant incontournable pour les primitives des architectures multiprocesseurs symétriques. On l'utilise avec des sections critiques de courte durée pour construire des mécanismes de plus haut niveau.

Ces mécanismes de contrôle servent aussi à bloquer passivement un processus qui doit attendre et ont alors des liens avec l'ordonnancement des processeurs. Ce sont les verrous des bases de données ou les sémaphores des systèmes d'exploitation. Dans les langages de programmation qui traitent la concurrence, les sémaphores ont été étendus par une structure modulaire appelée Moniteur (de Hoare ou de Brinch Hansen). Il leur correspond les méthodes synchronisées en Java et les objets protégés en Ada.

3. Les Sémaphores

3.1. Principe et réalisation des sémaphores

Un sémaphore S est un mécanisme auquel un processus n'a accès que par 3 opérations standard, ou primitives ou API ("application programming interface") qui sont appelées P(S), V(S) et E0(S, I). Ce mécanisme permet un autocontrôle sur la base d'un entier, qu'on peut interpréter comme un nombre d'autorisations (disponibles quand l'entier est positif, attendues quand l'entier est négatif). Il permet aussi de bloquer le processus demandeur (celui qui exécute P(S)) quand il n'y a plus d'autorisations disponibles et de réveiller un processus bloqué quand un autre processus rend une autorisation attendue (en exécutant V(S)). Il y a donc aussi une file d'attente de processus qui est associée à un sémaphore.

Quand un processus qui exécute P(S) doit être bloqué, le programme de la primitive P comprend la sauvegarde de son identité dans la file d'attente, la sauvegarde de son contexte puis l'appel à l'ordonnanceur de processus pour qu'un autre processus puisse utiliser le processeur. Quand le processus bloqué sera réveillé, il continuera son exécution en sortant de P(S) et en passant à l'instruction suivante dans son programme. Au niveau d'un processus, l'opération P(S) peut donc durer le temps de quelques instructions machine ou le temps nécessaire à la cohérence du système. Quand un processus bloqué est réveillé dans une primitive P(S) celle-ci inclut l'appel à l'ordonnanceur pour placer ce processus réveillé dans la file des processus prêts. Le choix du processus bloqué à réveiller est laissé à l'implantation. L'utilisation correcte des sémaphores ne doit pas dépendre d'une gestion particulière de la file d'attente. Il en est de même du choix à faire, dans une architecture monoprocesseur, entre le processus réveilleur (celui qui exécute V(S)) et le processus réveillé. Ils sont tous deux prêts et un seul peut être élu.

L'implantation d'un sémaphore S utilise en général une structure de données avec un entier noté S.E et une file d'attente notée S.F. On peut rencontrer d'autres implantation, comme une file unique pour l'ensemble des processus du système et un champ qui indique le nom du sémaphore qui bloque le processus.

Quoi qu'il en soit, l'aspect fondamental est que les sémaphores sont des ressources critiques et que leurs primitives sont des sections critiques exécutées de manière atomique. Pour les implanter dans un noyau de système monoprocesseur, on peut masquer et désactiver les interruptions pendant leur exécution. Dans un système multiprocesseur, il faut ajouter une attente active, comme on l'a vu en 2.9.

3.2. Exclusion mutuelle par sémaphore

On prend un sémaphore S qu'on initialise à 1, ce qui modélise une autorisation, celle d'entrer en section critique. L'entrée est alors contrôlée par une primitive P(S). Si plusieurs processus concurrents appellent P(S) en même temps, leurs exécutions de P(S) sont faites en séquence, une après l'autre. Le premier reçoit l'autorisation, après que l'exécution de P(S) ait décrémenté S.E et ramené sa valeur à 0. L'exécution suivante place S.E à - 1 et bloque le processus appelant. Etc... Quand le processus en section critique - le seul - quitte celle-ci, il doit rendre l'autorisation et réveiller l'un des processus bloqués. Cela est fait en appelant V(S). Le dernier processus à sortir de sa section critique et à appeler V(S) redonne, par l'exécution de V(S), à S.E la valeur 1.

La solution n'est pas toujours équitable. Elle l'est sur un système qui gère les files d'attente à l'ancienneté. À titre d'exercice, utiliser le schéma des sémaphores privés, présenté dans le chapitre suivant, pour donner une solution équitable quelle que soit la gestion des files d'attente. Donner ensuite une autre solution qui sert les processus en attente selon une priorité fixe quelle que soit la gestion des files d'attente.

3.3. Contraintes d'utilisation de l'exclusion mutuelle et des sémaphores

Les solutions supposent qu'un processus en section critique en sorte au bout d'un temps fini, donc qu'il n'y ait ni boucle infinie, ni blocage infini (d'où le danger de tout blocage en section critique, par un primitive P() en particulier), ni destruction de processus (si on ne le force pas à sortir avant sa destruction, ce sera infiniment sa dernière demeure!). Cette règle doit être

garantie par une certification statique du code ou par un contrôle dynamique, quel que soit le langage ou la méthode de programmation.

La programmation de l'exclusion mutuelle, telle que nous l'avons vue, suppose que l'on passe toujours par les procédures de contrôle ENTRÉE_SC et SORTIE_SC, donc qu'il n'y ait ni entrée ni sortie incontrôlées, y compris lors des erreurs, des exceptions, des trappes, et des destruction de processus.

Les effets des erreurs de programmation sont limités quand on utilise une programmation modulaire en
- encapsulant les données partagées à contrôler, avec les procédures d'accès et les mécanismes de contrôle (par exemple les sémaphores), comme en programmation par objets,
- distinguant l'interface d'appel (accessible par l'extérieur) et l'implantation (inaccessible par l'extérieur).

4. Programmation modulaire de l'exclusion mutuelle

4.1. Programmation automatique de l'exclusion mutuelle

Hoare et Brinch Hansen ont proposé dès 1972 d'introduire dans les langages de haut niveau les notions de région critique conditionnelle ou de moniteur et de leur faire correspondre automatiquement des exécutions en exclusion mutuelle.

Cette idée a été reprise dans la plupart des langages évolués qui gèrent des processus concurrents. En Java, les méthodes d'une classe peuvent être déclarées "synchronized" et ce sont des sections critiques pour l'accès à tout objet de la classe. En Ada 95, on peut déclarer un type ou un objet "protected" et toutes les procédures de ces objets protégés sont exécutées en exclusion mutuelle.

Malheureusement, le concept d'exclusion mutuelle ne suffit pas à traiter tous les paradigmes de la programmation concurrente. Il faut introduire en plus des mécanismes explicite de blocage et réveil, comme wait() et signal() pour les moniteurs, comme wait() et notify() en Java. Ada 95 introduit un blocage et un réveil implicites qui dépendent d'expressions booléennes appelées gardes et exprimées avec les variables de l'objet protégé. Cela permet de programmer un objet protégé comme un automate et cela fournit aujourd'hui la programmation de la concurrence qui est la meilleure et la plus fiable.

4.2. Programmation modulaire avec des sémaphores

Le sémaphore est un mécanisme qui permet le blocage et le réveil explicite. Il peut donc servir à traiter tous les paradigmes de la programmation concurrente. Mais son utilisation inconséquente peut être source de graves erreurs de programmation. Aussi pour obtenir une programmation fiable, nous adoptons la méthode de programmation suivante. Les données partagées à contrôler, leurs procédures d'accès et leurs sémaphores de contrôle sont encapsulés dans un paquetage Ada. On possède la modularité de la programmation par objets. On a une sécurité supplémentaire grâce à Ada qui distingue l'interface d'appel des procédures (accessible par l'extérieur) et le corps du paquetage ("package body", inaccessible par l'extérieur).

Ainsi pour l'exclusion mutuelle, on peut empaqueter la ressource critique et son sémaphore d'exclusion mutuelle et alors l'exclusion mutuelle devient semi-implicite car on appelle la procédure d'accès sans avoir à connaître ni sa programmation, ni ses sémaphores, et c'est dans le corps que l'on cache la programmation avec sémaphore.

On peut aussi déclarer les procédures de contrôle ENTRÉE_SC et SORTIE_SC dans l'interface d'un objet de contrôle, et cacher leur implantation, par sémaphore ou autre, dans le corps de cet objet. On reste avec une programmation explicite, mais on a restreint la portée des sémaphores au "package body" de l'objet de contrôle. On a limité les dégâts qu'aurait pu causer un sémaphore utilisé comme une variable commune à tous les processus, donc nécessairement globale au système.

4.3. Dualité processus et objets. Contrôle de concurrence, contrôle d'accès.

Dans tout système concurrent, on doit faire apparaître des activités concurrentes comme les processus ("threads", "tasks") et des objets partagés entre ces processus. Les processus

enchaînent des suites d'appels à des services ou à des objets. Les objets servent aussi à synchroniser les processus par rendez-vous, attente mutuelle ou synchronisation.

On a donc deux visions, celle du contrôle de concurrence et celle du contrôle d'accès. Toutes deux sont nécessaires et sont complémentaires.

Dans le contrôle de concurrence, les entités que l'on repère sont les processus. Cette vision se modélise bien en réseaux de Petri. On a une vision globale. Si on passe à la programmation en gardant cette globalité avec un accès global, on introduit le danger de propager globalement des erreurs. Toutefois ce niveau global reste nécessaire pour bien visualiser les relations entre processus et pour détecter les interblocages. Certains chercheurs appellent ce niveau celui de la concurrence inter-objets.

Dans le contrôle d'accès, les entités que l'on repère sont les objets partagés entre les processus. Cette vision se modélise bien avec un automate de transition d'états (compteurs, expressions de chemins, qu'on n'a pas présentés dans ce cours). Si on passe à la programmation, et en particulier à la programmation par objets, on doit contrôler leur accès et leur utilisation concurrente. Les mécanismes de synchronisation sont enfouis dans chaque objet. On obtient une programmation plus sûre, surtout si on sépare interface et corps de l'objet, comme le fait le langage Ada. Toutefois, avec cette approche, on a une vision locale et on a du mal à comprendre la solution globale et concurrente d'un problème. On risque aussi de ne pas repérer des interblocages. Certains chercheurs appellent ce niveau celui de la concurrence intra-objets.

4.4. Cas de la simulation avec le langage Ada

La norme du langage Ada autorise les compilateurs à optimiser l'utilisation des variables globales partagées (en dehors des tâches et des objets protégés) en en mettant une copie dans chaque tâche (dans la pile de la tâche, dans un cache ou dans un registre local, par exemple) et sans être obligé de garantir la cohérence de ces copies. La philosophie sous-jacente est que ces variables globales partagées doivent être des valeurs constantes. Pour simuler une variable globale modifiable, il faut se prémunir contre cette optimisation. On le fait en utilisant le pragma Atomic () ou le pragma Volatile(), qui empêchent l'optimisation par le compilateur.

Cela est nécessaire aussi quand les sémaphores sont simulés avec des objets protégés.

Par contre, quand les sémaphores sont simulés par des tâches et des rendez-vous de tâches, les pragma Atomic ou Volatil ne sont plus nécessaires. En effet les opérations P et V se font par simulation de rendez-vous. or un rendez-vous entre deux tâches est appelé un point de synchronisation et a pour effet de rendre cohérentes toutes les variables communes aux deux tâches (et aussi les paramètres passés entre tâches, quand ils sont passés par référence).

Exercice 8.1. Algorithme équitable avec TAS

L'algorithme donné en cours en 2.7 n'est pas équitable puisqu'un processus très rapide qui vient de sortir de section critique peut y entrer de nouveau avant tous les autres. Un algorithme équitable a été proposé par A. Burns en 1978 et est exposé dans le livre de Silberschatz et Gavin (Principes des systèmes d'exploitation, 4e édition, 1994).

Chaque processus j se déclare en attente et boucle tant qu'il est en attente et que la ressource critique est verrouillée. Tout processus j qui sort de section critique choisit un candidat k en attente s'il y en a et lui donne accès à la ressource critique, en écrivant qu'il n'est plus en attente. S'il n'y a pas de processus en attente, il déverrouille la ressource critique. L'équité est garantie par la manière de choisir k. On note les attentes dans un tableau de N variables (on doit savoir qu'il peut y avoir jusqu'à N processus) et on parcourt ce tableau de manière cyclique en commençant à j + 1 modulo N. Le parcours s'arrête quand on revient à i. (Notons que seul le processus qui est encore en section critique peut parcourir ce tableau. La cohérence du choix est garantie par cette unicité). Un processus k attend au plus N - 1 tours avant d'entrer en section critique.

Rédiger l'algorithme.

Solution de l'exercice 8.1.

```
function TAS(M : in out Boolean) return Boolean is
    Copie : Boolean;
begin Copie := M; M := True; return Copie; end TAS;

procedure TAS_equitable is  -- valable quel que soit le nombre de processus
                COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
                Verrou : Boolean := False;  -- initialisation du protocole d'exclusion mutuelle
                Attente : array (1..N) of Boolean := (others => False);

pragma Atomic(COMPTE_CLIENT, Verrou, Attente); -- directive au compilateur, voir 4.4
task P is                                                  •  task Q is
    X : Integer := 0;  -- variables locales à P            •      Y : Integer := 0;  -- locales à Q
    Ego : Mod N := UniqueDans(1..N);                       •      Ego : Mod N := UniqueDans(1..N);
    PasMoi : Boolean ;                                     •      PasMoi : Boolean ;
    K : Mod N;                                             •      K : Mod N;
begin                                                      •  begin
    for I in 1 .. 16 loop                                  •      for I in 1 .. 16 loop
        actions_hors _section_critique;                    •          actions_hors _section_critique;
        ----------------------------------------                      ----------------------------------------
        -- ENTREE_SC1 avec attente active                  •          -- ENTREE_SC2 avec attente active
        Attente(Ego) := True;                              •      Attente(Ego) := True;
        PasMoi := True;                                    •          PasMoi := True;
        While Attente(Ego) and PasMoi loop                 •      While Attente(Ego) and PasMoi loop
            PasMoi := TAS(Verrou) ;                        •              PasMoi := TAS(Verrou) ;
        end loop;                                          •          end loop;
        Attente(Ego) := False;                             •          Attente(Ego) := False;
        ----------------------------------------                      ----------------------------------------
            X := COMPTE_CLIENT;   -- P1                    •              Y := COMPTE_CLIENT; -- Q1
            X := X + 1;           -- P2                    •              Y := Y + 1;         -- Q2
            COMPTE_CLIENT := X;   -- P3                    •              COMPTE_CLIENT := Y; -- Q3
        ----------------------------------------                      ----------------------------------------
        -- SORTIE_SC1                                      •          -- SORTIE_SC2
        K := Ego + 1;                                      •          K := Ego + 1;
        While K /= Ego and (not Attente(K))                •      While K /= Ego and (not Attente(K))
            loop                                           •              loop
                K := K + 1 ; -- c'est modulo N             •                  K := K + 1;
            end loop;                                      •              end loop;
        if K /= Ego then Attente(K) := False;              •      if K /= Ego then Attente(K) := False;
                    else Verrou := false; end if;          •                  else Verrou := false; end if;
        ----------------------------------------                      ----------------------------------------
    end loop;                                              •      end loop;
end P;                                                     •  end Q;

begin null; end TAS_equitable;
```

Sûreté :
a) Le processus Ego n'a accès en section critique que si Attente(Ego) est False ou si PasMoi est False. Si plusieurs processus demandent la section critique alors qu'elle est libre, chacun met à True sa variable Attente. Comme PasMoi est sous le contrôle de TAS, un processus au plus (mais un au moins) obtient que PasMoi soit mis à False et entre en section critique.
b) Attente(Ego) a été écrite par le processus Ego et ne sera modifiée et mise à False que lorsque le processus qui exécute la section critique en sortira.
c) Tout processus qui sort de section critique, soit réveille un processus en attente, soit libère le verrou d'accès en section critique.
Vivacité : Le choix du prochain processus en attente se fait par un parcours cyclique du tableau Attente. Un processus attend au plus N -1 tours.