

## Régions Critiques.

Dans la programmation concurrente, le contrôle des accès aux objets partagés (par exemples les variables partageables, ...) par l'intermédiaire de l'outil sémaphore (précisément les primitives P et V) a pour objectif essentiel d'assurer l'intégrité et la cohérence de ces objets. Toutefois, bien que les sémaphores puissent être utilisés pour résoudre presque tous les problèmes de synchronisation, ils recèlent certains inconvénients qui peuvent dans certains s'avérer non négligeables.

Parmi ces inconvénients, on peut citer:

- Le manque de lisibilité des programmes, en particulier, lorsque la taille de ces derniers devient importante. L'absence de structuration des primitives P et V conduit à une localisation difficile des points de synchronisation (dans un programme).
- Leur sensibilité aux modifications: En effet, une modification dans un programme contenant des points de synchronisation (emplacements de P et V) peut conduire à réexaminer l'aspect synchronisation de la partie modifiée.
- Sensibilité aux erreurs accidentelles: En principe, l'exécution d'une section critique doit toujours commencer par une primitive P (utilisant un sémaphore de mutuelle exclusion) et terminer par la primitive V correspondante (sur le même sémaphore). La violation de ce principe, par un programmeur, peut conduire à une situation catastrophique. Cette situation peut provoquer: soit la violation de la section critique, les objets partagés ne sont donc plus protégés. Soit une incapacité de manipuler les objets partagés par suite d'inaccessibilité à la section critique (Une primitive P n'a pas de primitive V correspondante).

De même, sachant que les variables partagées doivent en principe être manipulées en exclusion mutuelle, c'est-à-dire dans des sections critiques, aucune obligation n'est faite au programmeur de respecter ce principe. Le non respect de ce principe, par inadvertance ou maladresse, risque de conduire à une incohérence. C'est pourquoi les régions critiques, dans leur notation structurée pour spécifier la synchronisation, ont été proposées pour alléger les différents inconvénients des sémaphores.

Le concept de régions critiques (Bri, Hoa), identique dans sa sémantique à celui des sections critiques (Dij), a été émis dans le souci de fournir au programmeur un outil capable de lui apporter une aide lors de la spécification de la synchronisation. Cette aide réside dans la garantie que toute variable déclarée explicitement partageable sera utilisée en exclusion mutuelle. Autrement dit, le mécanisme d'exécution des régions critiques veillerait à ce que ces variables soient manipulées correctement. Ce mécanisme, intégrable au compilateur, avertirait donc le programmeur de toute utilisation d'un objet déclaré partageable en dehors d'une région critique.

Les variables partagées sont explicitement placées dans des groupes appelés ressources. Chaque variable partagée peut être au plus dans une ressource. Par exemple, une ressource R contenant les variables V1, V2, ..., Vn est déclarée par: *Ressource* R: V1,V2, ..., Vn. Les variables de R peuvent être accessibles seulement à partir des formes structurales de régions critiques désignant ces variables.

Pour déclarer une variable partagée  $V$  du type  $T$  on utilise la notation suivante:

par exemple: *var* compte : *shared real*.

**Type** direction = *record* % Cf. exercice passage de véhicules sur un pont %

*end*

gauchedroit, droitgauche : direction;

*end*

On définit une région critique à l'aide des syntaxes suivantes:

### *Region V do Sc*

La séquence d'instructions  $Sc$  contient des variables déclarées partageables et éventuellement des variables locales au processus exécutant  $Sc$ .

## 2. Région critique conditionnelle

2

signifie: l'exécution de la région critique *sc* est conditionnée par la valeur de la condition *cond*. *Cond* représente une expression booléenne composée d'éléments de *V* et éventuellement de variables locales et constantes. Le scénario d'exécution de cette primitive par un processus appelant *P* est comme suit:

*P* accède à la région critique *sc* puis évalue *cond*, si *cond* est vraie, alors *P* exécute *sc* puis quitte *sc*.

sinon (*cond* est fausse) *P* sort de la région critique et se bloque jusqu'à ce qu'il soit réveillé lorsqu'un autre processus *Q* sort de sa propre région critique associée à *V*. *P* reprend alors l'évaluation de *cond* à son début.

#### **b) Region *V do sc await cond* ...**

Cette instruction combine les deux formes précédentes (2.7.1 et 2.7.2).

Le processus appelant exécute donc *sc* de façon inconditionnelle puis, pour poursuivre, se met en attente (se bloque) jusqu'à ce que la condition *cond* devienne vraie. Il est possible que *sc* soit une opération nulle auquel cas, l'évolution du processus appelant dépendra de la valeur de *cond* (attente si *cond* est fausse, poursuite en séquence sinon).

*Remarque:* Des régions critiques peuvent s'imbriquer à la manière des boucles *DO* des langages de programmation de haut niveau, toutefois une attention particulière peut leur être accordée afin d'éviter les interblocages éventuels. Par exemple, soient deux processus concurrents *P* et *Q* se partageant deux variables *v1* et *v2* tels que leurs codes soient:

*P:* Region *v1 do Region v2 do sc1;*

*Q:* Region *v2 do Region v1 do sc2;*

Dans pareil cas, il est évident de constater que si *P* et *Q* entrent simultanément dans leur propre région critique, un interblocage est inévitable. Il est possible de charger le compilateur qui gère les régions critiques de détecter des situations d'interblocage et d'en informer le programmeur pour agir en conséquence. Une action possible, et à titre indicatif, serait d'imposer un ordre d'utilisation des variables partagées (cf. exercice 14 a1)

Les programmes contenant des éléments de synchronisation spécifiés en termes de régions critiques sont plus lisibles. En effet, il est possible d'utiliser une approche axiomatique, basée sur la notion d'invariants, pour prouver la validité de l'exécution de la séquence d'instructions *Sc*. Ainsi, on peut associer un invariant *I<sub>v</sub>* à l'état de chaque ressource *V*, et un prédicat *P*. Le prédicat *P* ayant la valeur vraie à l'initialisation de *V* doit également avoir la valeur vraie à la fin de l'exécution de *Sc*.

Bien que le concept de régions critiques soit attractif, son implémentation s'avère pratiquement difficile et coûteuse. En effet, la condition *cond* contient des variables partagées (globales) et des variables locales au processus exécuté. Ces dernières imposent donc que l'évaluation de la condition *cond* soit faite par le processus exécuté. Ainsi, chaque processus

doit évaluer lui-même sa condition d'exécution de Sc, sans pour autant qu'il soit sûr de l'exécuter. On se trouve alors dans la situation où des processus sont réveillés systématiquement à la sortie de sc, puis éventuellement bloqués (cond est faux). Ces fréquentes commutations de contextes (sauvegarde et restauration des états), en particulier celles inutiles, constituent une source de gaspillage du temps processeur, notamment dans un système monoprocesseur. En revanche, elle peut être appliquée dans un système multiprocesseur en utilisant l'attente active.

Les régions critiques ont été implantées dans le langage Edison [Brin 1981] conçu spécialement pour des systèmes multiprocesseurs. Des variantes ont été également adaptées pour des environnements distribués [Bri, 1978; Lis 1982]

**Exercice:** Simuler un sémaphore à l'aide de régions critiques.

**var** sem: *shared integer* (sem  $\geq$  0);

**P(sem):** *region* sem **do** sem := sem - 1 *await* sem  $\geq$  0 ... (1) % incorrect %

Bien que la forme (1) précédente, traduisant textuellement la primitive P originelle, semblerait « correcte » indépendamment de V(sem), elle est à rejeter puisqu'elle ne répond pas à une association correcte avec la primitive de réveil V ci-après:

**V(sem):** *region* sem **do** sem := sem + 1;

En effet, l'exécution de V simulée précédemment a pour objectif de libérer un processus Q éventuellement en attente. D'après le principe des régions critiques, Q réveillé, doit évaluer sa condition d'entrer en section critique. Si celle-ci est vraie, il exécute effectivement sa section et poursuit en séquence; dans le cas contraire (condition fausse), il entre de nouveau dans la file d'attente.

Quand plusieurs processus se trouvent bloqués par P, impliquant sem  $<$  -1, l'exécution de V, à la suite de laquelle sem est incrémenté (sem := sem + 1) provoquerait des réévaluations des conditions de franchissement des points de synchronisation ineffectives, bien que logiquement, un franchissement au moins, devrait avoir lieu.

Par exemple, soient trois processus concurrents, P1, P2, P3, désireux d'entrer dans leur section critique respective (sem initialisé à 1), admettant que P1 soit le premier à exécuter la forme (1) précédente, sem devient = 0 et P1 accède à sa section critique (condition vraie). Si P2 exécute la forme (1) pendant que P1 est encore en section critique, sem devient = -1 et P2 se bloque (condition fausse). Si P3 exécute également (1), il se bloque car sem = -2. Lorsque P1 quitte sa section critique, il doit exécuter V(sem) qui incrémente sem (sem devient égal à -1). Le mécanisme d'exécution des régions critiques va réveiller P2 qui évalue sa condition de franchissement (sem  $\geq$  0) et trouve sem = -1 (condition fausse), P2 libère sa section critique et se bloque une nouvelle fois. P3 fait la même chose que P2, trouve sem = -1, et se bloque également. On voit bien que la section critique est libre et aucun des processus ne peut y accéder. P1, lui-même, ou tout autre nouveau processus concurrent ne peut accéder à sa section critique, il se produit donc un blocage. Ce problème est dû au fait que la

décrémentation de  $\text{sem}$  s'est faite avant le test de la condition de franchissement. La solution correcte est donc la suivante:

**Region** sem **when** sem > 0 **do** sem := sem - 1 ...

**Exercice:** Donnez une implémentation des régions critiques conditionnelles à l'aide des sémaphores.

Soit la structure **Region V when** cond **do** sc:

L'entrée dans la section critique `sc` qui manipule la variable partagée `V` doit être préservée par un sémaphore de mutuelle exclusion *mutex* de valeur initiale 1. L'attente de la condition `Cond` se fait sur le sémaphore *attente* de valeur nulle; ainsi:

```
Var mutex: sémaphore init 1,      % sémaphore de mutuelle exclusion %
    attente: sémaphore init 0,
    nbattente: integer   init 0;   % nombre de processus en attente d'entrée dans sc %
    nbreveil : integre   init 0;   % "                "         réveillés           %
```

**P(mutex);**

**if not cond then begin**

$$\text{nbattente} := \text{nbattente} + 1;$$

**V** (mutex);

**P(attente);**

**While not cond do begin**    % réveil de tous les processus en attente%

```
nbreveil := nbreveil + 1;
```

**if** nbreveil < nbattente

**then  $V(\text{attente})$**

**else** V(mutex);

**P(attente);**

**end;**
$$\text{nbattente} := \text{nbattente} - 1;$$

**end:**

SC;

**if** nbattente > 0 **then begin**

```
nbreveil := 0;
```

**V(attente):**

**end;**

**else** V(mutex);