

## **Communication interprocessus**

Les ressources dans un système, quelles soient matérielles (hard) ou logicielles (soft) doivent, si elles sont utilisées de manière efficace, être partagées; tel est d'ailleurs un des objectifs assignés au système d'exploitation. Par exemple, un canal est capable de servir les requêtes d'E/S reçues de façon plus rapide que le processus qui les génère. Aussi, n'y a-t-il pas meilleure utilisation que de faire partager un canal unique entre plusieurs processus en servant toutes les requêtes émises par ces derniers.

De même, le partage du logiciel est non moins important pour accroître l'efficacité. Ainsi, le partage d'un compilateur permettrait d'optimiser l'espace mémoire en ayant une copie unique sollicitée lors des compilations (intérêt de la réentrance); car dans le cas contraire, tout processus qui lance une compilation serait contraint de charger dans son propre espace une copie du compilateur. Autrement dit, on aurait autant de copies du compilateur que de compilations à effectuer. Donc, le partage de ressources tant matérielles que logicielles améliore grandement l'efficacité du système.

Il faut noter que ces deux catégories de ressources possèdent des propriétés communes; notamment un processus utilisant une ressource ignore si celle-ci est à caractère logiciel (implémentée par logiciel) ou physique (implémentée par matériel). C'est le cas, où pour permettre le partage d'un disque entre plusieurs processus, le système implémente un certain nombre de disques logiques (virtuels) qui simule chacun le disque physique. Chaque disque logique est un processus qui communique avec le canal du disque réel.

Relativement au partage, il n'y a en général pas de distinction entre ressources logiques et ressources physiques, leurs comportements externes sont équivalents. Bien que ce partage tire meilleur parti de l'utilisation des ressources, toutefois cet usage doit s'effectuer à bon escient dans le respect de la cohérence et l'intégrité. Pour pouvoir atteindre cet objectif, il est impératif que les processus impliqués dans ce partage puissent communiquer entre eux à volonté et au besoin de se synchroniser.

## **Synchronisation des processus**

Un système opérationnel a pour objectif essentiel de rendre des services aux usagers; et plusieurs processus de ce système peuvent collaborer à la réalisation de ces services. Cette collaboration induit une concurrence relativement à l'accès aux objets partagés. Il en résultent alors des relations d'ordre (lors de ces accès) qui, pour des raisons logiques, imposent une exécution cadencée de certaines opérations. Par exemple, suite à un événement, un processus est amené à se bloquer, à en bloquer ou à en réveiller d'autres. Autrement dit, l'évolution de chacun de ces processus concurrents est assujettie à des règles qui fixent sa propre cadence d'exécution. On désigne ainsi par *synchronisation* cette cadence de déroulement des processus qui imposent un ordonnancement à des endroits bien déterminés de leurs codes, appelés

*points de synchronisation.* Les règles de synchronisation peuvent être par exemple liées à la précedence, la priorité, ou l'exclusion mutuelle dans le temps.

Remarque: En dehors des points de synchronisation, les processus concurrents évoluent librement (sans contraintes de synchronisation) et indépendamment les uns des autres...

Pour pouvoir appliquer cette synchronisation aux processus concurrents, il y a lieu d'utiliser les mécanismes appropriés. Ces derniers sont soit fournis sur appel direct au système (dans le noyau), soit incorporés dans les langages de programmation utilisés.

On peut classer les mécanismes de synchronisation en deux catégories:

- Ceux qui permettent d'agir directement sur le processus à synchroniser par l'intermédiaire de son identificateur (nom processus) qui est fourni comme paramètre; on les appelle mécanismes de *synchronisation à action directe*.

Par exemple:

*Bloquer*(P:nom\_processus) signifie: état(P):= bloqué, retirer à P le contrôle du processeur.

*Réveiller*(Q:nom\_processus) signifie: état(Q):= prêt, éventuellement allouer le processeur au processus Q.

- Ceux qui servent d'intermédiaire à travers lesquels, la synchronisation déclenchée par un processus, agissent sur d'autres processus concurrents. Ils sont identifiés par mécanismes de synchronisation à *action indirecte*, par exemple: le mécanisme des verrous, sémaphores, moniteurs, expressions de chemins, module de contrôle etc.

Bien que les ressources physiques et logiques à un seul point d'accès puissent être partagées, elles sont généralement utilisées par un processus à la fois. Une ressource ne pouvant être allouée qu'à un seul usager à la fois est appelée *ressource critique*.

Par exemple: une imprimante est une ressource physique critique (non partageable réellement ou physiquement). De même, une variable globale est une ressource logique non partageable donc critique.

Dans le cas où plusieurs processus souhaiteraient le partage de l'usage de telle ressource (critique), ils doivent synchroniser leurs opérations de sorte qu'au plus un processus ait le contrôle de celle-ci. Si la ressource est utilisée par un processus, tous les autres processus concurrents ou compétitifs qui la désirent s'excluent temporairement. Ils doivent attendre sa libération pour pouvoir l'utiliser.

Dans chaque processus, les sections (séquences) d'instructions à travers lesquelles se fait l'accès à ces ressources critiques peuvent être isolées. Ces sections ou régions, appelées *sections critiques*, doivent avoir la propriété d'être *mutuellement exclusives*. Autrement dit, à tout instant un processus au plus peut être en exécution dans une telle section critique. Une fois son exécution lancée, une section critique doit être fiable (éviter les blocages), et aussi brève que possible pour éviter une longue attente aux processus qui la demandent (permettre une plus grande simultanéité).

Beaucoup de ressources physiques telles que les unités de bandes magnétiques, les scanners etc., constituent des ressources critiques. Les variables partagées pouvant être mises à jour par plusieurs processus sont également des ressources critiques.

Pour illustrer cette notion, considérons par exemple deux processus Débit et Crédit se partageant une variable commune nommée Compte. Si Débit et Crédit tentent de mettre à jour simultanément Compte, la valeur finale peut donner lieu à une erreur. On parlera alors de l'*incohérence* de la valeur de Compte.

Supposons explicitement que Compte représente le compte bancaire (ou CCP) d'un client. Que Débit est le processus déclenché à chaque opération de retrait effectué sur Compte, et Crédit le processus inverse qui réalise l'opération d'ajout.

Admettons que Compte doit subir simultanément un crédit d'un montant C et un débit d'un montant D, on peut alors avoir le scénario suivant lors des exécutions:

" Processus Crédit "

" Processus Débit "

A:  $\text{Compte} := \text{Compte} + C$

B:  $\text{Compte} := \text{compte} - D$

En décomposant les actions A et B ci-dessus, en actions élémentaires, on obtient:

|                                      |                                      |                           |
|--------------------------------------|--------------------------------------|---------------------------|
| a1: $\text{Rla} := \text{Compte}$ ;  | b1: $\text{Rlb} := \text{Compte}$ ;  | % Lecture de compte %     |
| a2: $\text{Rla} := \text{Rla} + C$ ; | b2: $\text{Rlb} := \text{Rlb} - D$ ; |                           |
| a3: $\text{Compte} := \text{Rla}$ ;  | b3: $\text{Compte} := \text{Rlb}$ ;  | % mise à jour de compte % |

Rla (respectivement Rlb) représente un registre ou variable locale de Crédit (respectivement Débit). La valeur finale de Compte peut être incohérente, notamment si les opérations s'effectuent selon certains ordres par exemple:

a1, b1, a2, b2, a3, b3 (1) donnerait pour valeur finale erronée de Compte qui serait: valeur initiale moins D au lieu de valeur initiale plus C moins D.

De même, la séquence d'exécution b1, a1, b2, a2, b3, a3 (2) donnerait à Compte: la valeur initiale plus C.

En fait, l'incohérence (l'inexactitude) des résultats produits par les exécutions des opérations précédentes est une conséquence de la perte de mise de la variable globale non partageable *compte*. Dans le premier cas (1), on a perdu la première mise à jour de compte ( $\text{compte} + D$ ) effacée (écrasée) par la deuxième mise à jour  $\text{compte} - D$ . Dans le second cas (2), il s'est produit l'inverse, où la première mise à jour de compte par  $(\text{Compte} - D)$  a été effacée par la deuxième mise à jour  $(\text{compte} + D)$ . Seule la dernière mise à jour a été effectuée !

Pour éviter de pareils résultats inattendus (incohérents), la mise à jour de Compte (ressource critique) doit être protégée donc doit se faire dans une section critique. On va aborder ce problème dans la suite en tenant compte de l'évolution historique des mécanismes.

### 1 Solution logicielle (théorique).

La question que l'on peut se poser peut être la suivante:

Comment peut-on protéger une section critique par des moyens purement logiciels, en se basant seulement sur l'indivisibilité des accès mémoire? C'est-à-dire que le matériel décide de l'arbitrage en cas de conflits lors des accès simultanés à une variable en mémoire. Autrement dit, le matériel établit un ordonnancement lors des accès concurrents par plusieurs processus aux cellules mémoire.

Compte tenu de cette hypothèse, Dekker répondit le premier à la question en considérant deux processus (la généralisation pour N processus est donnée dans Dijkstra 68).

Nous allons, dans ce qui suit, tenter plusieurs approches pour réaliser l'exclusion mutuelle de deux processus cycliques doté chacun d'une section critique. Cette démarche par approches successives revêt un caractère pédagogique, et montre un raisonnement évolutif pour arriver enfin à une solution finale correcte.

a) Comme première tentative de réalisation de l'exclusion mutuelle de deux processus concurrents, on va supposer que les sections critiques sont protégées par une variable "barrière" prenant deux valeurs possibles "ouverte" ou "fermée".

Si la valeur de barrière est fermée, cela signifie qu'un processus est entrée dans sa section critique; dans le cas contraire (barrière = ouverte), les sections critiques sont libres et un processus peut en obtenir l'accès. Une "solution" possible est:

```

Var barrière : (fermée, ouverte)    % variable partagée %
    Barrière := ouverte                % initialement la section critique %
                                      % est libre %

```

" **Processus1** "

Test: *Tantque* barrière = fermée *faire*

    allera test

*Fintantque*

    barrière := fermée;

    < section critique >;

    barrière := ouverte;

" **Processus2** "

Test: *Tantque* barrière = fermée *faire*

    allera test

***Fintantque***

```
barrière := fermée;
< section critique >;
barrière := ouverte;
```

En examinant ces algorithmes, on a l'impression que l'exclusion mutuelle est garantie. Cependant, en communiquant via la variable partagée Barrière, les processus peuvent trouver Barrière ouverte au même instant, et entreront simultanément chacun dans sa section critique (cf. décomposition précédente en processus Debit et Credit). Finalement, la communication au moyen d'une variable commune unique révèle une insuffisance pour garantir l'exclusion mutuelle.

b) Considérons maintenant l'usage d'une variable *Tour* pour établir un ordre d'entrée en sections critiques. *Tour* = 1 quand le processus1 peut entrer dans sa section critique, et *tour* = 2 quand il sera possible au processus2 d'entrer lui aussi dans sa propre section critique. Ainsi on peut préconiser la "solution" suivante:

```
Var Tour = 1..2 ;      % variable partagée %
      Tour := 1 ;      % initialisation %
```

" **Processus1** "

" **Processus2** "

Test: ***Tantque*** tour=2 ***faire***

allera test

***Fintantque***

< section critique >;

Tour := 2;

Test: ***Tantque*** tour=1 ***faire***

allera test

***Fintantque***

< section critique >;

Tour := 1;

La solution garantit effectivement l'exclusion mutuelle mais au prix d'une contrainte sévère inacceptable: Les processus exécutent leurs sections critiques dans l'ordre suivant 1, 2, 1, 2 etc., de plus, s'il y a arrêt ou régression de vitesse du processus1, il y va de même pour le processus2, d'où le rejet de la solution.

c) Cette alternance d'accès en section critique peut être évitée en accordant à chaque processus sa propre variable locale de valeurs possibles "intérieur" ou "extérieur". Intérieur indique que le processus désire entrer, ou bien est déjà dans sa section critique. Extérieur : signifie que le processus est à l'extérieur de sa section critique.

Chaque processus examinera donc la variable de son partenaire avant de pénétrer dans sa propre section critique, d'où l'algorithme suivant:

```

Var process1, process2 : (intérieur, extérieur);
    process1 := extérieur;      % initialisation %
    process2 := extérieur;      %      "      "      %

```

**" processus1 "**

Process1 := interieur;

test: **Tantque** process2 = intérieur **faire**

*allera* test

**Fintantque**

        < section critique >;

        process1 := extérieur;

**" processus2 "**

Process2 := interieur;

test: **Tantque** process1 = intérieur **faire**

*allera* test

**Fintantque**

        < section critique >;

        process2 := extérieur;

Dans la solution ci-dessus, il n'y a pas de variable partagée et l'arrêt d'un processus en dehors de sa section critique n'affecte aucunement la progression du processus concurrent partenaire. Cependant, une nouvelle difficulté apparaît.

En effet, si les deux processus font simultanément les affectations (process1 := intérieur, process2 := intérieur), alors ils entreront dans une boucle infinie (chacun attend l'action entreprise par l'autre).

**d)** La boucle infinie détectée précédemment provient du fait qu'en cas de conflit chacun des processus attend son partenaire pour prendre l'initiative de l'action. Si tout processus qui détecte un conflit (les deux processus essayent simultanément d'entrer dans leurs sections critiques respectives) change sa valeur, le problème pourrait être résolu. La solution à l'aide d'une telle stratégie est la suivante:

```

Var process1, process2 : (intérieur, extérieur);
    process1 := extérieur;      % initialisation %
    process2 := extérieur;

```

**" Processus1 "**

**Tantque** process1 = extérieur **faire**

    process1 := intérieur;

```

    si process2 = intérieur alors
        process1 := extérieur;
test: Tantque process2 = intérieur faire
    allera test
    Fintantque
Fintantque
    < section critique >;
    process1 := extérieur;

```

" Processus2 "

*Répéter*

```

    process2:=intérieur;
    si process1 = intérieur alors
        process2 = extérieur;
    Répéter jusqu'à process1 = extérieur
jusqu'à process2 = intérieur
    < section critique >;
    process2 = extérieur

```

Malheureusement la solution ci-dessus n'est pas encore correcte; elle peut conduire à un état de blocage si l'évolution temporelle des deux processus est exactement identique (cette situation de blocage est similaire à celle de c)).

e) Dekker fut le premier à pouvoir présenter une solution correcte exempte des problèmes rencontrés précédemment. Sa solution consiste essentiellement en une combinaison des deux approches précédentes, I.e.. chaque processus a son propre indicateur, indiquant s'il désire entrer dans sa section critique, et un entier permettant de résoudre le conflit lorsque les deux processus décident simultanément d'accéder à leurs sections critiques.

### Solution correcte

```

Var process1, process2 : (intérieur, extérieur)
    tour : 1..2;
    process1 := extérieur; % initialisation %
    process2 := extérieur;
"Processus1"
    Process1 := intérieur;
    si process2 = intérieur alors
        debut
        si tour = 2 alors debut
            process1 := extérieur

```

```

        Répéter jusqu'à tour = 1
        process1 := intérieur
    fin
    Répéter jusqu'à process2 = extérieur
fin
< section critique >;
    tour := 2;
    process1 := extérieur;

```

### "Processus2"

```

process2 := intérieur;
si process1 = intérieur alors
    debut
        si tour = 1 alors debut
            process2 := extérieur;
            répéter jusqu'à tour = 2
            process2 := intérieur;
        fin
        Répéter jusqu'à process1 = extérieur
    fin
    < section critique >;
    tour := 1;
    process2 := extérieur;
Exclusion mutuelle: solution de Dekker

```

### Variante de la solution de Dekker

La solution précédente est quelque peu compliquée et manque relativement de lisibilité, on peut lui préférer la solution plus simple (compacte) suivante due à Peterson [Peterson 81.]

```

Var process1, process2 :(interieur, exterieur);
    tour : 1..2;
    process1:= exterieur;
    process2:= exterieur;

« Processus1 »
process1:= interieur;
tour:= 2;
test: tantque process2 = interieur et tour = 2 faire
    aller à test

```



```

    Fintantque
    <section critique>
process1:= exterieur;

« Processus2 »
process2 := interieur;
tour := 1;
test: tantque process1 = interieur et tour = 1 faire
    allerà test
    Fintantque
    <section critique>;
process2:=exterieur;

```

Remarque: La version de l'algorithme précédent pour n processus concurrents se trouve dans Peterson [Peterson et al.].

La méthode précédente de réalisation de l'exclusion mutuelle est plutôt encombrante et impraticable. Cependant, toute technique ou mécanisme d'élaboration de l'exclusion mutuelle doit fournir une solution devant vérifier les propriétés suivantes à savoir:

1. A tout instant, un processus au plus peut se trouver dans sa section critique;
2. Le blocage d'un processus hors de sa section critique ne doit en aucun cas affecter l'évolution des autres processus partenaires (concurrents);
3. Aucune hypothèse n'est faite sur la vitesse d'exécution des processus (celle-ci est quelconque);
4. Tout processus qui désire entrer dans sa section critique pourra le faire au bout d'un temps fini.

L'examen détaillé de la solution logicielle précédente a permis de mettre en évidence le problème de l'exclusion mutuelle d'un point de vue théorique.

En pratique, des méthodes beaucoup plus efficaces ont été proposées et certaines d'entre elles ont été et continuent d'être appliquées; ces dernières feront l'objet d'une présentation dans les sections suivantes.

## 2 Masquage des interruptions

La première solution au problème de réalisation de l'exclusion mutuelle peut être accomplie par le matériel. Lorsqu'un processeur unique est employé (système monoprocesseur), la seule cause possible d'exécution imbriquée ou alternée de séquences d'instructions appartenant à divers processus est l'*interruption*. Donc, si chaque fois qu'un processus désire entrer dans sa section critique, il masque (ou inhibe) les interruptions et les démasque (les autorisent) à la

sortie, alors l'exclusion mutuelle entre processus compétitifs ou concurrents est garantie. Bien entendu, cette garantie n'est plus assurée dans le cas d'un système multiprocesseurs (car le masquage s'applique uniquement aux interruptions d'un processeur donné).

En effet, chaque processeur dispose de son propre répertoire d'instructions en l'occurrence les instructions privilégiées de masquage et démasquage des interruptions qui nous intéressent particulièrement. L'effet de ces instructions ne s'applique qu'au processeur qui les exécute, comme le montre la figure suivante:

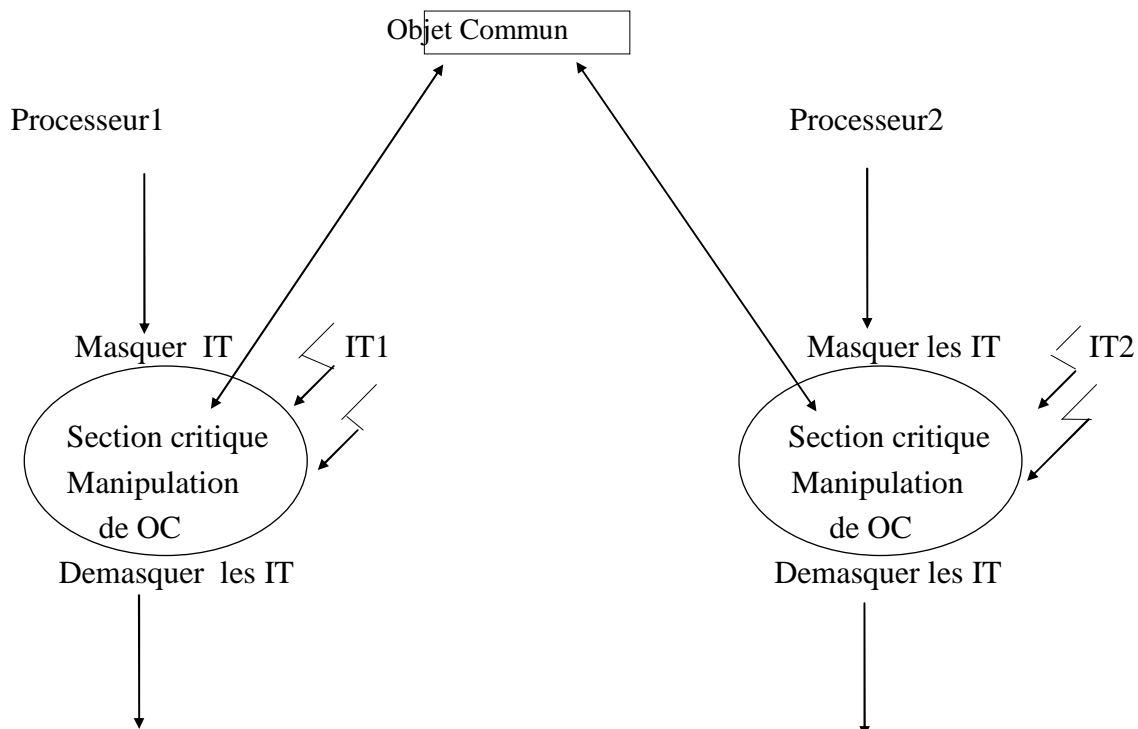


Figure 3. Inadéquation de la technique du masquage/démasquage en multiprocesseur

IT1: Interruptions du processeur1

IT2: Interruptions du processeur2

Quand un processeur  $P_i$  veut entrer en section critique, il masque au préalable ces interruptions  $IT_i$ . Ce masquage ne s'applique qu'aux  $IT_i$  de ce processeur  $P_i$ ; cela n'empêche donc pas le processeur  $P_j$  d'entrer dans sa propre section critique et de manipuler l'objet commun OC; ce qui violerait l'exclusivité de l'accès à l'objet OC.

Par exemple, en reprenant les processus Débit et Crédit précédent, on peut écrire:

Débit

:

Crédit

:

|                         |                         |
|-------------------------|-------------------------|
| ·                       | ·                       |
| STM                     | STM                     |
| Compte := compte + vald | Compte := compte - valc |
| CLM                     | CLM                     |
| ·                       | ·                       |
| ·                       | ·                       |
| ·                       | ·                       |

où STM: SeT Mask: masquer les interruptions,  
CLM: CLear Mask : démasquer les interruptions.

### Avantages et inconvénients du Masquage/Démasquage

- Comme tout processeur possède, dans son répertoire d'instructions, des instructions de masquage et démasquage de ses interruptions, et ces instructions agissent au niveau matériel, il suffit de réaliser un *protocole d'entrée* (ou prologue d'entrer) en section critique à l'aide de l'instruction de masquage et un *protocole de sortie* (ou épilogue) correspondant à l'aide de l'instruction de démasquage.

Ainsi, l'avantage majeur d'une telle technique réside dans sa simplicité et son efficacité.

- Toutefois, le pouvoir de l'inhibition (masquage) peut s'étendre sur tout le système sans possibilité de sélection, d'importantes parties du système peuvent alors se voir différer l'usage des interruptions, particulièrement lorsque les sections critiques nécessitent un temps d'exécution important. Donc, afin d'éviter de pénaliser temporellement le système lors de la prise en compte immédiate des événements externes (associés aux interruptions), il est recommandé d'utiliser prudemment d'une telle technique.

### Attente Active

La solution logicielle de l'exclusion mutuelle (voir problème de Dekker précédent) se base fondamentalement sur le concept d'indivisibilité des accès mémoire; cependant si deux opérations particulières peuvent être exécutées de façon *atomique* tel que: a) tester et modifier, ou b) échanger deux valeurs, alors l'exclusion mutuelle peut être programmée beaucoup plus aisément.

#### a) Test And Set (TAS)

Soit C une variable globale indiquant l'état (ou l'occupation) d'une section critique, c'est-à-dire:  $C = 1 \implies$  section critique occupée, et  $C = 0 \implies$  section critique libre. L'algorithme de fonctionnement de TAS est le suivant:

```
TAS(C)
  begin
    < verrouiller l'accès à C >;
```

```

lire C
if C = 0 then begin
    C := 1
    co := co + 2    % co = compteur ordinal % (1)
    end            % ou compteur d'instructions %
    else co := co + 1    % (2) %
endif
    < libérer l'accès à C >
end

```

(1): Signifie: Branchement à la première instruction de la section critique et exécution

(2): Signifie: Branchement à l'instruction qui suit immédiatement l'instruction TAS. Cette instruction sera nécessairement un branchement vers l'instruction TAS elle-même.

L'emploi de TAS se fait comme suit:

Soit C une variable matérialisant l'état d'une ressource critique R:

C = 0 ==> R est libre, C = 1 ==> R occupée.

```

    C := 0          % initialisation %
Test: TAS(C);      % prologue %
    allera Test
    < section critique >
    C := 0          % épilogue %

```

Par exemple, la programmation de l'exemple précédent peut être:

| debit                 | Credit                |
|-----------------------|-----------------------|
| .                     | .                     |
| .                     | .                     |
| .                     | .                     |
| Test: <b>TAS</b> (C); | Test: <b>TAS</b> (C); |
| allera Test;          | allera Test;          |
| a1: Rla := Compte ;   | b1: Rlb := Compte ;   |
| a2: Rla := Rla + C ;  | b1: Rlb := Rlb - D ;  |
| a3: Compte := Rla ;   | b3: Compte := Rlb ;   |
| C := 0;               | C := 0;               |

## b) Instruction Exchange

Elle permet d'échanger les valeurs de deux emplacements mémoire de manière indivisible.

Soit C une variable globale partagée initialisée à 0;

```

Var C : 0..1
    C := 0          % initialisation: si C = 0 alors la ressource est libre, occupée sinon %
"Processus P"
Var local : 0..1;    % variable locale au processus P %
    local := 1;
While local = 1 do

```

```

    EXCHANGE (local, C)
  endwhile;
  < section critique >;
  C := 0;

```

où l'algorithme de EXCHANGE est:

```

EXCHANGE (local, C):      % opération indivisible %
  begin
    Temp := local          % Temp: variable ou registre intermédiaire %
    local := C;
    C := Temp;
  end
Exclusion mutuelle utilisant l'instruction EXCHANGE

```

La technique d'attente active a), b), est acceptable si la demande en ressource(s) critique(s) est faible et les sections critiques courtes; elle devient inadéquate dans un système monoprocesseur.

En effet, prenons par exemple un système monoprocesseur avec deux processus P1 et P2. Supposant que le système est en temps partagé et que P1 est en cours d'exécution. P1 exécute TAS et entre dans sa section critique SP1 et commence son exécution. Le quantum de temps de P1 étant épuisé (P1 n'est pas encore sorti de SP1), le système alloue le processeur à P2 qui teste l'entrée de sa section critique SP2, Il boucle en attendant que P1 libère sa section critique. P2 bouclera dans TAS jusqu'à épuisement de son quantum. Après quoi, le système lancera P1 qui continuera l'exécution de SP1. P2 pourra entrer dans sa section critique SP2 lorsque P1 aura quitté sa section critique SP1. Une ou plusieurs commutations de contexte de P2 auraient été effectuées inutilement sans que P2 ait pu entrer dans SP2(gaspillage du temps processeur).

Supposons maintenant que l'ordonnancement des processus est basé sur des priorités et que P2 est plus prioritaire que P1. Lorsque P1 est dans sa section critique SP1, un évènement de réveil de P2 se produit faisant passer celui-ci à l'état prêt. Immédiatement P2, plus prioritaire que P1, est lancé. P2 teste son entrée dans sa section critique SP2, il *boucle indéfiniment* sur le TAS car SP1 est toujours occupée par P1 qui est en attente.

Toutefois, bien que cette technique consomme du temps processeur, le temps d'attente ne constitue guère un overhead substantiel. Pour éviter ces attentes actives consommatrices de temps processeur, des mécanismes moins coûteux en temps sont utilisés.

### Verrous

Pour éviter la pénalité temporelle de l'attente active, il est préférable d'utiliser des méthodes plus efficaces permettant la mise en file d'attente d'un processus demandeur d'entrée en

section critique lorsque celle-ci n'est pas libre. Le contrôle du processeur est alors affecté à un autre processus prêt. Le verrou et les primitives qui le manipulent constituent une de ces méthodes d'attente passive.

Un verrou est une cellule mémoire ou variable qu'on note  $V$  et à laquelle on associe une file d'attente  $f(V)$ . On utilise les verrous pour réaliser des sections critiques. L'emploi de verrou  $V$  se fait au moyen de deux primitives: *verrouiller*( $V$ ) et *deverrouiller*( $V$ ) comme suit:

Initialement  $V$  est nul et  $f(V)$  est vide;

*Verrouiller*( $V$ ): **debut**

**si**  $V = 0$  **alors**  $V := 1$ ;

**sinon debut**

mettre le processus appelant  $P$  dans  $f(V)$

$\text{état}(P) := \text{bloqué}$ ;      % voir figure 1. %

**fin**

**finsi**

**fin**

*Deverrouiller*( $V$ ): **debut**

**si**  $f(V) \neq \text{vide}$  **alors debut**

sortir  $Q$  de  $f(V)$ ;

$\text{état}(Q) := \text{prêt}$ ;

**fin**

**sinon**  $V := 0$ ;

**finsi**

**fin**

Donc, un processus qui ne peut entrer en section critique ( $V=1$ ) entre dans la file d'attente  $f(V)$ . Lorsqu'un processus quitte sa section critique, il doit exécuter la primitive *Deverrouiller*( $V$ ) (épilogue), alors un des processus en attente dans  $f(V)$  est activé (si  $f(V) \neq \text{vide}$ ).

Le verrou  $V$  et sa file d'attente  $f(V)$  constituent des objets partagés (pour divers processus) qu'il faut protéger pour assurer leur intégrité. C'est pourquoi, leur manipulation doit se faire dans une section critique. Les deux opérations *Verrouiller*( $V$ ) et *Deverrouiller*( $V$ ) doivent donc se comporter, vis-à-vis des processus appelants, comme deux primitives: c'est-à-dire deux opérations indivisibles. Cette indivisibilité peut se réaliser de deux manières:

- Par le mécanisme simple de masquage et démasquage des interruptions, si le système est monoprocesseur. On considère ainsi le processeur comme ressource critique.
- Puisque le masquage/démasquage ne peut s'appliquer qu'aux interruptions d'un processeur donné, un autre processeur, exécutant un processus concurrent, peut violer cette

indivisibilité. L'usage de l'instruction TAS ou d'une instruction équivalente, dans le cas d'un système multiprocesseurs, est donc bien adapté.

*Remarque:* Il faut noter la différence importante entre l'usage de TAS, pour protéger des sections critiques d'une part, et rendre les opérations verrouiller et déverrouiller indivisibles d'autre part.

Dans le premier cas, l'attente active produite par TAS dure le temps d'exécution d'une section critique. Ce temps, dépendant étroitement de la longueur de la section critique, peut constituer un facteur de performance non négligeable.

Dans le second cas, l'attente active générée par TAS ne dure que le temps d'exécution des instructions de l'opération verrouiller ou déverrouiller. Ce temps est en général insignifiant.

Les verrous peuvent être considérés comme un cas particulier des sémaphores qui eux constituent un outil beaucoup plus général.