République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Ferhat Abbas-Sétif 1

جامعة فرحات عباس - سطيف1

Faculté des Sciences
Département d'Informatique

Polycopié de la matière

SYSTEME D'EXPLOITATION 2
PROCESSUS CONCURRENTS

Cours

Destiné aux étudiants 3ème année licence LMD

Pr Zibouda ALIOUAT

2023-2024

1. Introduction

Les traitements simultanés (ou encore processus concurrents, activités parallèles) constituent une des caractéristiques essentielles des systèmes d'exploitation. Cette simultanéité peut s'observer au niveau matériel (processeurs centraux, canaux, périphériques, ...) à partir duquel est établie celle du niveau logiciel. Son importance s'inscrit dans les objectifs assumés par les fonctions d'un système d'exploitation opérationnel, afin de pouvoir tirer meilleur parti des possibilités matérielles, notamment l'augmentation de la puissance des traitements. Cette puissance que l'on s'efforce d'accroître sans cesse par divers moyens (matériels/logiciels) est à la base d'un objectif primordial de réalisation d'un temps de réponse optimal.

Le but de ce fascicule est de donner un certain nombre de concepts fondamentaux relatifs à l'unité de base de l'activité d'un système d'exploitation (le processus), d'étudier les propriétés intrinsèques et extrinsèques qui régissent cette unité, et surtout d'examiner son comportement environnemental (interactions) vis-à-vis de ses pairs au sein d'un système informatique.

Une question qui vient à l'esprit est: pourquoi est-on amené à définir et utiliser une entité relativement moins usitée ou moins "familière" que le programme ?

En effet, le programme est probablement la notion la plus employée par les usagers d'un système informatique pour solliciter les services de ce dernier. Toutefois, cette notion s'avère insuffisante pour observer de façon adéquate la dynamique qui s'opère dans un système. Cette insuffisance est principalement due au fait qu'un programme peut être constitué d'un ensemble de procédures, et une procédure unique peut donner naissance à plusieurs exécutions. Cette multiplicité d'exécutions découle particulièrement de la notion de réentrance des procédures. Une procédure est dite réentrante si elle peut satisfaire plusieurs appels simultanés; ce qui impose qu'elle soit écrite en code pur c'est-à-dire, non modifiable en cours d'exécution (c'est en général le cas). Elle doit également et impérativement manipuler plusieurs ensembles de données, chacun d'eux correspondant à un appel distinct. Autrement dit, elle doit travailler sur des données localisées sur l'espace de son appelant: elle ne dispose donc pas de variables locales propres.

Lorsqu'une procédure n'est pas réentrante, son exécution est en relation bijective avec son invocation (à toute exécution correspond un appel et un seul). Toutefois, l'intérêt que revêt la réentrance (gain important d'espace mémoire et temps) impose aux systèmes modernes de doter systématiquement les procédures de la capacité de réentrance (Bien que la réentrance dépende des possibilités du jeu d'instructions d'une machine, elle constitue néanmoins la clé de la multiprogrammation dans des systèmes tel que Unix).

Ainsi, la correspondance biunivoque procédure $\longleftrightarrow$ exécution unique ne répond plus à une description appropriée d'un système d'exploitation actif. C'est pourquoi une entité dynamique, mieux adaptée: le *processus*, est nécessaire pour modéliser l'activité dans un tel système.

Notons néanmoins que dans certaines situations, la réentrance n'est pas souhaitable car elle va à l'encontre de l'objectif d'optimisation de la ressource mémoire centrale. C'est notamment le cas, par exemple, d'un utilisateur qui, n'ayant pas reçu immédiatement de réponse à sa commande d'exécution de programme, s'évertuerait à relancer ce dernier par plusieurs frappes successives. Ces dernières provoqueraient autant d'exécutions du même programme alors qu'une seule exécution est souhaitée d'où gaspillage de la ressource mémoire et processeur.

2. Processus

2.1 Définitions

La notion de *processus* (ou tâche) est largement employée lorsqu'il s'agit de considérer un système d'exploitation; toutefois, le formalisme qui lui y est consacré n'est point à la mesure de l'importance et de la fréquence de son utilisation. Cependant, les auteurs s'accordent à mettre en évidence le caractère dynamique de ce concept abstrait qui modélise les composants d'un système. Ainsi, un processus peut être considéré comme l'activité qui découle de l'exécution d'un programme. Selon Dijkstra (68), un processus est défini comme ce qui peut arriver lors de l'exécution d'un programme séquentiel. Une définition formelle de cette notion se trouve dans Hor(73). Dans ce qui suit, et sauf indication contraire, nous considérerons un processus comme une entité dynamique résultant de l'exécution d'une procédure séquentielle. Un programme, quant à lui, peut être composé d'une ou plusieurs procédures, et une procédure est formée d'une suite d'instructions. Une procédure constitue une entité statique, et l'exécution d'une instruction génère une action élémentaire; de ce fait, un processus se présente comme une succession d'actions élémentaires (une action elle-même peut être considérée comme un microprocessus). Pour des raisons de performance, liées particulièrement au coût temporel de la commutation de contexte, certains systèmes considèrent une notions beaucoup plus fine que le processus à savoir: le *thread*.
Un thread est un processus léger. Ainsi, un processus classique peut être composé de plusieurs threads se partageant un même espace mémoire que le processus père.

Pour qu'un programme puisse donner naissance à un ou plusieurs processus, il est indispensable de disposer d'un agent initiateur. Cet agent est un dispositif réalisé partiellement ou totalement de composants matériels. On désigne communément l'agent exécuteur sous le vocable de *processeur*. Donc, un processeur est un mécanisme matériel ou association matériel/logiciel qui active un processus en exécutant la procédure associée. Désormais, on dira qu'un processeur "exécute" un processus.

2.1.1 Etats d'un processus

En général, quand un processus est créé (par une primitive du système au profit d'un autre processus) celui-ci n'obtient pas immédiatement le contrôle du processeur. Il se met alors dans un certain état dit: état *prêt* (ready). Au moment opportun, et suite à un événement bien défini, il pourra alors entrer dans l'état *actif* (courant, running) et acquérir ainsi le

contrôle du processeur désiré. Autrement dit, des événements liés à la disponibilité des ressources font évoluer un processus d'état en état jusqu'à sa terminaison. Le cycle de vie d'un processus se résume donc en une suite de transitions d'états comme le montre la figure 1 ci-après.

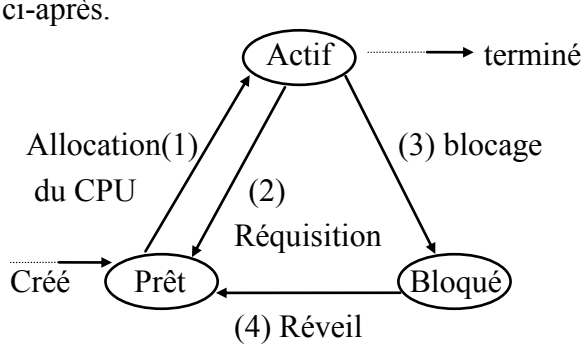


Figure 1. Etats d'un processus

La transition (2): Réquisition (preemption): consiste en le retrait du processeur au processus actif, pour le compte d'un autre processus prêt, bien que le premier en a encore besoin. Cette transition peut avoir lieu, par exemple, lorsque le processus actif perd le contrôle du processeur qu'il détient, suite à l'épuisement du quantum de temps qui lui a été alloué. Ce peut être également l'action du *dispatcher* (ordonnanceur des processus) qui réquisitionne le processeur au profit d'un autre processus plus prioritaire.

Les transitions d'états (1) et (2) sont donc liées à l'allocation du processeur physique aux processus demandeurs, alors que les transitions (3) et (4) sont essentiellement dues aux attentes d'évènements tels que: la synchronisation des processus, les attentes d'entrés/sorties, d'émissions et récéptions de messages, etc.

Les états principaux: *actif*, *prêt*, *bloqué* (attente) peuvent traiter un grand nombre de situations possibles. Toutefois, certaines applications peuvent nécessiter une vision des états beaucoup plus fine; par exemple, pour permettre la mise au point d'un programme, il peut s'avérer utile d'introduire un état supplémentaire *suspendu*. Ainsi, l'usager en interaction (via un debugger) avec son programme (ou plutôt avec le ou les processus généré(s)), et afin d'examiner l'évolution de ce dernier, peut agir de la manière suivante: Régulièrement suspendre le processus pour examiner la validité des résultats intermédiaires élaborés. Au vu de ces résultats, l'usager décidera de faire évoluer son processus dans l'état prêt (résultats intermédiaires satisfaits) ou au contraire procéder à son arrêt (ou terminaison).

2.1.2 Contexte d'un processus

Pour être actif, un processus a besoin d'un environnement dans lequel il s'exécute. Cet environnement est constitué de l'ensemble des informations (ou objets) manipulables lors de son exécution et désigne son *contexte*; ce contexte comprend en particulier:

a) Des informations qui définissent l'état du processeur

On y trouve les registres généraux accessibles aux programmes utilisateurs, et des registres internes d'un usage spécial réservé. Parmi ces derniers, un registre particulier identifié par mot d'état du processeur ou PSW (Program Status Word) contient les informations concernant:

- *Etat d'exécution*: indication sur l'activité ou l'attente du processeur.

Remarque: L'état « attente » considéré peut être un état d'attente active dans lequel le processeur exécute une boucle sans fin en attendant l'arrivée d'une interruption le faisant basculer dans l'état actif effectif.

- *Mode d'exécution*

Il existe plusieurs modes d'exécution d'un programme suivant que l'usager ait ou non accès à certains privilèges du système. Cette notion de privilège résulte du souci de protéger mutuellement les usagers entre eux et de pouvoir protéger également le système vis-à-vis de ces derniers.

Selon les systèmes, on distingue deux ou plusieurs modes d'exécution. On peut en trouver au minimum deux modes: un mode système (ou superviseur) le plus privilégié et un mode utilisateur : le moins privilégié. Il est possible de subdiviser d'avantage ces modes afin d'offrir aux processus juste ce qui leur faut comme privilèges. Par exemple, les systèmes VAX de Digital offrent quatre modes d'exécution: Le mode noyau (le plus privilégié), le mode superviseur, le mode exécutif, et le mode utilisateur (le moins privilégié).

- *Masquage des interruptions*:

Pour des raisons de monopole du processeur (par un processus) durant l'exécution d'une séquence d'instructions, notamment pour rendre celle-ci indivisible ou *atomique*, il importe que le processeur soit actif avec interruptions *inhibées* ou masquées. Autrement dit, l'arrivée d'un signal d'interruption peut provoquer la réaffectation du processeur au profit d'un autre processus au détriment du processus courant. Les problèmes résultant de cette commutation seront abordés dans le chapitre suivant relatif à la synchronisation.

- *Informations relatives au processus courant*:

On y trouve en particulier: des indicateurs de protection de l'espace mémoire accessible, les droits d'accès à celui-ci, le code condition, le compteur ordinal, etc...

b) L'espace mémoire:

Cet espace mémoire comprend: Le segment de code (ou segment procédure), le ou les segment (s) de données, la pile d'exécution, une zone de sauvegarde des registres en cas d'interruptions, et les zones de communication avec d'autres processus, etc...

Exemple: Le contexte d'un processus dans le système d'exploitation VAX/VMS se compose de deux parties: Un contexte matériel et un contexte logiciel.

Le contexte matériel comprend:

1. **Le *PSL*** (Processor Status Longword) qui constitue le double mot d'état du processeur, il contient diverses informations telles que: les codes conditions, les bits d'interruptions, les modes d'accès, etc ...

2. **Le compteur ordinal ou PC** (Program Counter) matérialisé par le registre R15, qui contient l'adresse de l'instruction à exécuter lors de l'interruption.
3. **Les registres généraux**: R0 à R14,
4. **D'autres registres spéciaux** (associés aux périphériques, gestion de la mémoire, gestion des processus, ...).

Le contexte logiciel comprend:
- L'identification du processus, son propriétaire,
- Sa priorité, ses privilèges,
- Les ressources allouées, du type: limites et quota du temps processeur, de l'espace mémoire et disque, les périphériques accessibles, etc.

Remarque: Le contexte d'un processus est maintenu dans une structure de données appelée PCB (Process Control Bloc). Tous les PCB's des processus sont regroupés pour former des listes (listes: de processus prêts, bloqués) manipulées par le système.

Une application est généralement composée d'une multitude de processus travaillant en coopération et un système peut gérer simultanément plusieurs applications différentes; quelles relations peut-il exister donc entre divers processus dans un système ?

2.2 Gestion hiérrarchique de processus

2.2.1 Classes de processus

Généralement on distingue trois catégories de processus dans un système:

- *Processus indépendants*

Des processus sont dits indépendants si les ensembles d'objets auxquels ils accèdent sont disjoints c'est-à-dire: l'activité de chacun est privée (isolée) de celles des autres. La forme la plus simple d'indépendance s'observe quand les processus sont physiquement séparés, autrement dit, il y a absence d'interaction mutuelle (cas où chaque processus s'exécute sur un processeur non connecté à d'autres); cela revient à dire que les contextes des processus ne sont constitués que d'éléments privés.

- *Processus coopérants*

De tels processus ont accès aux objets partagés utilisés directement comme supports de communication interprocessus. La circulation d'informations qui implique une interdépendance est généralement une caractéristique des systèmes transactionnels; les transactions dans ce type de systèmes constituent normalement un ensemble de processus coopérants (chaque transaction est un processus du système). En général, chacun de ces processus effectue un service propre qui entre dans le cadre d'une contribution à la réalisation d'une tâche globale, autrement dit, ils coopèrent à la finalisation de cette tâche.

- *Processus compétitifs*

Lorsqu'isl existent dans un système, des objets "rares" à usage privé (non partageables), dont l'accès est sollicité simultanément par plusieurs processus, ces derniers sont appelés processus compétitifs. Ainsi, la coopération induit une relation de compétition.

2.2.2 Création et destruction de processus

2.2.2.1 Création

Deux approches sont généralement utilisées dans les systèmes pour créer des processus. La première consiste à considérer un nombre fixe de processus créés au moment de la génération du système, et le lancement d'une activité revient donc à obtenir le contrôle d'un des processus déjà existant. Cette création statique est adoptée dans des environnements peu évolutifs où les caractéristiques des applications sont bien connues, par exemple, un système de contrôle de processus industriels (systèmes de conduite automatique d'engins, systèmes embarqués, ...).

Dans un environnement moins bien caractérisé où le nombre de processus est imprévisible et évolue dans le temps, notamment dans les systèmes à usage général (systèmes interactifs), la seconde approche, dite de création dynamique, s'impose de manière impérative. Dans la suite, et pour sa généralité, on adoptera cette seconde approche.

La création d'un processus se traduit donc comme un service accompli (sur demande) par le système au profit d'un autre processus. Le demandeur peut être soit un processus utilisateur soit un processus interne du système.

L'opération de création consiste d'abord en la mise en place de l'environnement associé au processus à créer (création de son contexte et initialisation), puis le cas échéant activer celui-ci. Bien entendu, la requête de création doit fournir au système certains paramètres utiles tels que le nom du processus à créer, et éventuellement certains attributs du type: priorité, privilèges, ressources, etc... Les paramètres nécessaires mais non obligatoires qui ne sont pas fournis se verront attribués, par système, des valeurs par défaut. Un nombre minimal de paramètres est donc impératif, par exemple le nom du processus à créer.

Schématiquement, l'opération de création peut se décomposer en plusieurs étapes:

1. Acquisition, au profit d'un processus en cours de création, dans l'espace du système, d'un bloc de contexte.
2. Localisation, à partir du nom du programme associé au processus, du fichier correspondant (à noter que le nom du processus dans le système est différent de celui du programme associé).
3. A partir de ce fichier (localisé par le système de gestion de fichiers) qui constitue une image-mémoire produite par l'éditeur de liens, on peut obtenir certaines informations

nécessaires pour l'exécution ultérieure telles que: la taille du programme, l'adresse de lancement, etc.
4. Après l'acquisition des informations en 3. et relativement à la méthode de gestion de la mémoire utilisée dans le système, il y aura détermination d'un espace mémoire libre où sera chargé le programme. Après quoi, ce dernier subit un transfert physique de la mémoire auxiliaire (disque) vers la zone de la mémoire centrale qui lui est réservée.
5. Mise à jour du bloc contexte associé, puis mise en file d'attente des processus prêts.

A noter que le processus nouvellement créé n'est pas nécessairement activé. Il le sera ultérieurement après exécution du dispatcher (répartiteur). Ce dernier décidera de l'octroi du contrôle du processeur soit au processus créateur, soit au processus créé ou à un autre processus de plus grande priorité. Ainsi la procédure de création de processus se termine sur un appel du dispatcher.

Il existe, entre le processus créateur (ayant demandé la création) soit P1 et le processus créé soit P2, une *relation de filiation*: P2 est appelé processus *fils* de P1 qui en est le *père*. Cette relation père-fils pouvant être récursive (P2 appartenant à la descendance de P1 peut être père à son tour), peut être généralisée en donnant lieu à une arborescence de processus comme l'illustre la figure 2. ci-après.
Bien entendu, cette relation de filiation est présente dans le bloc contexte de chaque processus (ascendant ou descendant) à l'aide d'un lien vers le bloc contexte du père ou du fils. L'importance de filiation réside dans le pouvoir qu'elle attribue au père pour contrôler le(s) fils; notamment le pouvoir de communication ou de destruction. A noter que l'intersection des contextes du créateur et du processus créé n'est pas vide.
Par exemple, dans le système Unix, tous les processus (sauf le bootstrap) sont créés dynamiquement par la primitive *Fork* comme suit:
Process-ident := Fork(). Avec process-ident: indentification du processus à créer, entier ≥0
L'exécution de cette primitive provoque la création d'un processus fils. Les deux processus, le père et le fils ont des contextes identiques. Ils ne se distinguent l'un de l'autre que par leur identité (valeur fournie par la primitive fork) comme indiqué ci-après.

```
 Programme
int:  process-ident
          ⋮
  Process-ident := Fork(.)

si  process-ident = 0 alors  debut
                                séquence de code exécutée ;  exec(p)  (*)
                                par le processus fils
                             fin
sinon   si   process-ident = nil alors
                                 < traiter cas d'erreur > ; création impossible
```

sinon debut
séquence de code exécutée par le processus père
fin
finsi
finsi

Un processus père P crée un processus Q pour attendre un service de ce dernier. P dispose d'une primitive wait lui permettant d'attendre la fin de l'exécution de l'un de ses fils. La synchronisation entre les processus Père et fils se limite donc à l'usage de la primitive système Wait, qui suspend l'exécution du processus Père jusqu'à la fin d'exécution de l'un de ses fils:

Process-ident := Wait(état) où état représente l'état de terminaison.

La valeur de process-ident renvoyée par la primitive Wait permet au père de connaître l'identité du processus fils dont l'exécution s'est achevée.

Un processus met fin à son exécution par la primitive *Exit* .

(*): La primitive Exec(p) permet à un processus de changer de contexte, en appelant une procédure spécifiée P.

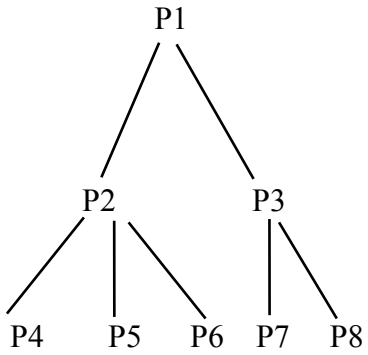


Figure 2. Structure arborescente de processus

2.2.2.2 Destruction

La destruction d'un processus Pi est réalisée par le système au profit d'un processus Pj doté du privilège de destruction. En général, le demandeur Pj est un processus père. Toutefois, Pj peut être un processus système ou un autre processus privilégié (éventuellement à l'initiative de Pi lui même, quand il atteint une fin normale).

Lorsqu'un processus à détruire possède une descendance (fils), il est courant (mais non impératif) de détruire d'abord cette descendance (fils, petits fils ...). Par exemple, la destruction de P3 demandée par P1 peut provoquer la destruction au préalable de P7 et P8.

Il faut remarquer que la destruction d'un processus intervient à la fin normale de l'exécution du programme correspondant (fin normale). Il se peut que cette destruction puisse être demandée après constat d'une anomalie dans le fonctionnement d'un processus

(boucle infinie, résultats fournis partiellement erronés, suite à un interblocage, ...) on parlera alors de fin anormale du processus.

Dans tous les cas de destruction, le système doit effectuer les opérations suivantes:

1. Libérer les ressources allouées (physiques et/ou logiques) au processus par exemple: espace mémoire, périphériques, fichiers utilisés, verrous, section critique, ... etc.
2. Libérer le bloc contexte alloué au processus,
3. Réallouer le processeur à un processus prêt (branchement au dispatcher).

Remarque: Pour qu'un processus, ayant atteint sa fin normale, puisse être détruit par le système (sauf si le processus doit être maintenu résident en mémoire), il est nécessaire que ce dernier en soit informé. C'est en fait le processus lui-même qui fera appel au système lui indiquant que son exécution est terminée. Ce qui signifie que la dernière «instruction » d'un programme doit être un appel système sollicitant implicitement cette destruction.

2.3 Communication interprocessus

Les ressources dans un système, quelles soient matérielles (hard) ou logicielles (soft) doivent, si elles sont utilisées de manière efficace, être partagées. 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 (à cause 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.

2.4 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 précédence, 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'accés 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). 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: Compte:=Compte + C B: Compte:= compte - D

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

```
a1:  Rla := Compte  ;        b1:  Rlb := Compte  ;   % Lecture de compte %
a2:  Rla := Rla + C ;        b2:  Rlb := Rlb - D ;
a3:  Compte := Rla  ;        b3:  Compte := 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 le première mise à jour de compte (compte+D) effacée par la deuxième mise à jour compte - D. Dans le second cas (2), il s'est produit l'inverse, où la première mise à jour de compte par (Compte - D) a été effacée par la deuxième mise à jour (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.

2.4.1 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.

2.4.2 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:

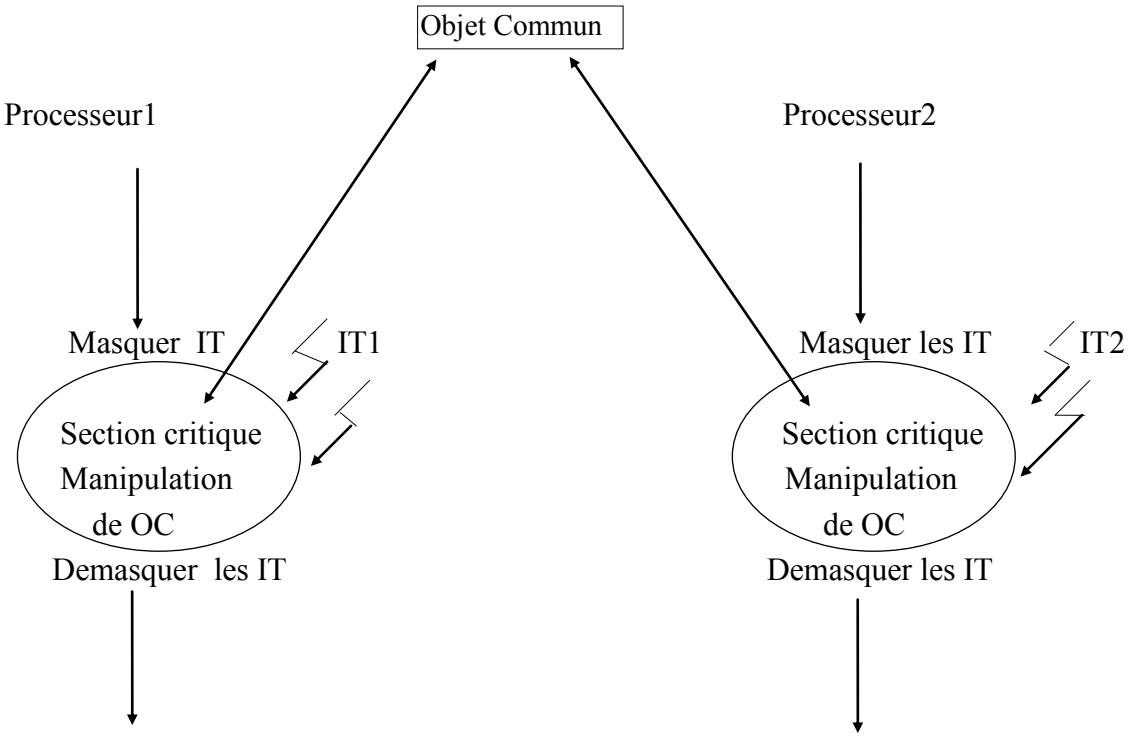


Figure 3. Inadéquation de la technique du masquage/démasque en multiprocesseurs

IT1: Interruptions du processeur1
IT2: Interruptions du processeur2

Quand un $processeur_i$ veut entrer en section critique, il masque au préalable ces interruptions ITi. Ce masquage ne s'applique qu'aux ITi de ce $processeur_i$; cela n'empêche donc pas le $processeur_{3-i}$ d'entrer dans sa propre section critique et de manipuler l'objet commun OC; ce qui violerait l'exclusivité de l'accès à 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.

2.4.2.1 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'user prudemment d'une telle technique.

L'accroissement de la capacité des traitements et les exigences en matière de fiabilité de ces derniers (acquérir une plus grande disponibilité matérielle) militent en faveur de disposer d'installation multiprocesseur. Dans pareilles situations, la technique du masquage/démasquage si simple et si efficace ne peut être appliquée. Donc l'inconvénient premier du masquage et démasquage est d'être inutilisable dans un environnement multiprocesseurs.

2.4.3 Attente Active

La solution logicielle de l'exclusion mutuelle 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 ==> section critique occupée, et C = 0 ==> 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 le TAS. Cette instruction sera nécessairement un branchement vers l'instruction TAS.

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: TAS(C);                      Test: TAS(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 inadmissible dans un système monoprocesseur. Toutefois, dans tous les cas, 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.

2.4.4 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)
```

```
            état(P) := bloqué;          % voir figure 1. %
          fin
      finsi
   fin
```

```
Deverrouiller(V): debut
          si f(V) ≠ vide alors  debut
                                  sortir Q de f(V);
                                  état(Q) := 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) ≠ 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.

<u>*Remarque*</u>: 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 *Deverrouiller* 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.

3. Sémaphores

3.1 Introduction

Les caractéristiques communes des techniques de réalisation de sections critiques présentées dans la section précédente se situent soit dans le gaspillage (ou le monopole) du temps processeur induit par l'attente d'entrer en section critique, soit dans leur difficulté à être appliquées de manière générale pour des problèmes plus compliqués. Il serait donc préférable d'une part, qu'un processus qui ne puisse avoir accès à la section critique (celle-ci étant occupée) puisse se voir retirer le contrôle du processeur pour se mettre à l'état dormant (attente passive ou bloqué) sans gaspillage du temps processeur. Ce dernier sera réveillé lorsque l'accès à sa section critique devient possible. Et d'autre part, de disposer d'un mécanisme suffisamment général pour être aisément applicable dans des problèmes dotés d'une certaine complexité. C'est cette idée de base qui orienta Dijkstra à introduire la notion de sémaphore. Cette notion fournira alors une généralité permettant de réaliser l'exclusion mutuelle des sections critiques et jouer le rôle de compteur de ressources. Son adaptation à une grande variété de problèmes, mêmes complexes, fera du sémaphore un outil de référence.

Plutôt que de présenter la notion de sémaphore de mutuelle exclusion qui n'est qu'un cas particulier, on donne une description globale de sémaphore général. Ce concept de sémaphore fournit un moyen naturel de gestion de ressources (comptabilité, allocation des ressources, synchronisation des processus).

3.2 Notion de sémaphore

Un sémaphore (noté sem) est constitué d'un doublet [Csem, Qsem] ou Csem représente le compteur du sémaphore *sem* à valeur initiale positive ou nulle; et Qsem la file d'attente associée à sem initialement vide. Habituellement, on manipule le sémaphore sem comme une variable du type sémaphore plutôt que son compteur Csem.

Un sémaphore ne peut être manipulé autrement que par les primitives associées notées P (ou Wait) et V (ou Signal) ci-après.

3.2.1 Fonctionnement de P et V

Soit un sémaphore sem, l'algorithme de la primitive P ou wait, eventuellement bloquante peut s'écrire comme suit:

```
P(sem)
 begin
  Csem := Csem - 1
  if Csem < 0 then begin                    % P = processus appelant %
                     état(P) := bloqué;
                     mettre P dans Qsem;
                   end
  endif
```

end.

L'algorithme de la primitive V ou Signal s'écrit comme suit:

```
V(sem)
 begin
   Csem := Csem + 1;
   if Csem ≤ 0 then begin
                       < sortir un processus Q de Qsem >;
                        état(Q) := prêt;
                      end
   endif
 end.
```

<u>Remarque</u>: Il est possible de considérer le compteur d'un sémaphore comme ayant toujours une valeur entière positive. Toutefois, les algorithmes des primitives P et V doivent correspondre à cette nouvelle définition:

```
P(sem):
  begin  if  csem > 0 then csem := csem -1
         else  begin
                  état(p) := bloqué;
                  mettre P dans Qsem;
               end;
          endif
end;
V(sem):
  begin
    if  Qsem ≠ vide then  begin
                             < sortir un processus Q de Qsem >,
                             état(Q) := prêt
                          end
    else csem := csem + 1
  end;
```

Dans la première définition, quand la valeur du compteur du sémaphore *csem* est négative, sa valeur absolue indique le nombre de processus en attente dans la file *Qsem*. Par contre, dans la seconde définition, il faut tester Qsem pour déterminer le nombre de processus bloqué(s) sur le sémaphore sem.

Bien que par abus de langage ou pour simplicité, on désigne un sémaphore par un entier seulement, l'implémentation réelle dans un système impose d'associer à cet entier un pointeur vers sa file d'attente associée.
Dans la suite, sauf indication contraire, on adoptera la première définition. On peut utiliser la structure d'implémentation suivante:

```
Type  semaphore record
                    csem : integer  init val
                    qsem: queue
                  end
```

3.2.2 Caractéristiques de P et V

Les primitives P et V sont ininterruptibles et peuvent faire partie du noyau du système. Elles s'excluent mutuellement, autrement dit à tout instant au plus une d'entre elles peut être en exécution. Elles sont implémentées soit par le matériel (câblage ou micro-programmation) soit par logiciel (programmation). La primitive P peut être bloquante: I.e.. le processus appelant P est mis en file d'attente (cas ou Csem < 0) à la suite de quoi, le processeur est réalloué à un processus de la file des processus prêts à exécution. Au contraire, la primitive V est passante (non bloquante) et peut éventuellement faire sortir un processus Q de Qsem et le rendre à l'état prêt.

La stratégie selon laquelle est gérée la file Qsem est déterminée lors de l'implémentation. Elle doit être choisie suivant l'objectif pour lequel le sémaphore correspondant est utilisé.
En général, le noyau peut adopter la stratégie FCFS (First-Come Fist-Served) pour sortir de Qsem le processus ayant le plus longuement attendu (stratégie équitable). Il peut également choisir un algorithme plus complexe tel celui basé sur la priorité des processus lorsque le temps d'attente dans Qsem devient significatif.

C'est ainsi qu'il est judicieux d'attribuer une priorité (dynamique) fonction du nombre de ressources coûteuses contrôlées par un processus entrant dans Qsem. Puisque sitôt ce dernier sera réveillé et réactivé, sitôt il terminera son exécution et libérera les ressources coûteuses occupées.

3.2.3 Utilisation de sémaphores

Dans un cas tout à fait général, un sémaphore de valeur initiale quelconque positive ou nulle (par exemple N) peut être utilisé pour synchroniser l'accès des processus concurrents à une ressource partageable à N points d'accès. Cette ressource peut donc être allouée simultanément à N processus. Le sémaphore est alors considéré comme représentant le nombre de copies de la ressource.
Lorsqu'une ressource commune R n'a qu'un seul point d'accès (N=1:un seul exemplaire) autrement dit ressource critique, l'usage d'un sémaphore à valeur initiale unitaire permet de réaliser l'exclusion mutuelle lors des accès à R.
Donc pour réaliser l'exclusivité d'exécution d'une section de programme : I.e. créer des sections critiques, il y a lieu de faire usage de sémaphores à valeur initiale égale à 1. Ces

sémaphores portent le nom de sémaphore d'exclusion mutuelle. Aussi, toute section de programme deviendra *section critique* en l'encadrant par *P(mutex)* à l'entrée de la section (prologue), et *V(mutex)* à la sortie (épilogue) avec bien entendu mutex: sémaphore initialisé à 1.

Exemple : ***Var*** mutex: *semaphore*;

```
mutex := 1;

P(mutex)            % prologue %

< section critique >;

V(mutex)            % épilogue %
```

Lorsque la valeur initiale d'un sémaphore est nulle (0), celui-ci est désigné par *sémaphore privé*. Il permet, en particulier, au processus détenteur de ce sémaphore de se bloquer en attente d'un événement particulier. Cet événement peut être: une communication synchrone de message, une attente de la fin d'une entrée/sortie, etc.

Par exemple, soient deux processus coopérants P1 et P2 agissant comme suit:

```
Var semprive1: semaphore init 0;
    semprive2: semaphore init 0;
      P1                          P2

  Recevoir (P2, msg);          Emettre (P1, msg);
  P(semprive1);                V(semprive1);

Emettre (P2, msg);           Recevoir (P1, msg);
  V(sempriv2);                 P(semprive2);
```

Avec Emettre (Pi, msg): Transmettre un message *msg* au processus Pi,
Recevoir(Pj, msg): Recevoir du processus Pj un message *msg*.

La réception de message est ici bloquante, de sorte que le processus Pi qui demande un message se bloque sur son sémaphore privé [**P**($semprive_i$)] jusqu'à ce qu'il soit réveillé par le processus émetteur Pj en exécutant **V**($semprive_i$).

La même situation se produit lorsqu'un processus effectue une E/S synchrone. L'attente de cette E/S se fait par l'intermédiaire de l'exécution d'une opération P sur un sémaphore

privé. Le réveil peut être réalisé par l'exécution de l'opération V sur ce même sémaphore privé, dans la routine d'interruption associée à la fin de l'E/S.
Un sémaphore de valeur initiale nulle (sémaphore privé) ou unitaire (sémaphore de mutuelle exclusion) porte le nom de sémaphore *binaire*.

L'outil sémaphore est intrinsèquement général et complet mais conduit souvent à une structuration de programme difficile à comprendre (peu lisible) et sensible à des modifications. Autrement dit, la dispersion des primitives P et V à travers le code d'un programme, particulièrement lorsque celui-ci est de taille non négligeable et comprenant un nombre important de points de synchronisation, rend sa lecture (compréhension) fastidieuse. De même, une modification (ajout ou retrait) d'une partie de code renfermant un ou plusieurs points de synchronisation peut mettre en cause la validité de la partie contrôle (synchronisation) du programme tout entier!
L'inconvénient premier se situe notamment au niveau de la programmation qui requiert du programmeur une attention particulière; en l'occurrence, un oubli d'une primitive V (correspondante à P) ou l'écriture de P à la place de V conduit directement à un blocage infini.

3.2.4 Implémentation des primitives P et V

Les algorithmes de P et V précédents doivent être implémentés en tant que primitives. Ces primitives, qui ne sont que des procédures, doivent obligatoirement être dotées du caractère d'atomicité (indivisibilité) au niveau de leur exécution. P et V peuvent être utilisées pour réaliser des sections critiques et elles-mêmes doivent être considérées comme des sections critiques atomiques. Cette atomicité peut être réalisée à l'aide des mécanismes matriels vus précédemment (TAS, Exchange, Instructions de masquage/démasquage des interruptions, ...). Ainsi, si le système considéré est doté d'un processeur unique, le masquage/démasquage des interruptions peut facilement convenir. Dans le cas d'un système multiprocesseurs, on peut utiliser l'instruction TAS. Cependant, si tel est le cas, l'attente active ne dure que le temps d'exécution de P ou V qui est en général très court (comme dans le cas des primitives verrouiller et déverouiller précédentes)..

3.3 Problèmes possibles de l'exclusion mutuelle

Comme mentionné précédemment, l'exclusion mutuelle est indispensable pour protéger et garantir la cohérence des ressources critiques dans un système. Toutefois deux problèmes cruciaux peuvent surgir lors de la programmation de sections critiques: ce sont l'*interblocage*(deadlock) et la privation (starvation) ou famine .

3.3.1 L'Interblocage

L'interblocage (ou deadlock), comme l'illustre la figure 4. ci-après, génère une situation problématique ou plusieurs processus concurrents pour des ressources communes non partageables se bloquent mutuellement.

Cet état résulte du fait que chacun des processus (en concurrence) est en possession d'une ou plusieurs ressources réclamée(s) par d'autres et réciproquement. En d'autres termes, les processus concurrents se trouvent bloqués par manque de ressources, et l'événement de déblocage ne peut provenir que de l'un d'entre eux (déjà bloqué) ou par intervention extérieure forcée (système d'exploitation, opérateur humain).
Plus généralement, un interblocage de processus concurrents se produit dans un système si et seulement les quatres conditions suivantes se réalisent conjointement, à savoir:
1- *Exclusion mutuelle*: Demandes concurrentes d'utilisation de ressource(s) non partageable(s). Autrement dit, il existe au moins une ressource demandée dont l'utilisation doit être en exclusion mutuelle,
2- *Allocation et attente*: Un des processus concurrents est en possession d'au moins une ressource non partageable et attend l'allocation d'autres ressources détenues par d'autres processus,
3- *Attente circulaire*: Il existe dans le système un ensemble de processus en attente {P0, P1, ..., Pn} de sorte que P0 attende une ressource détenue par P1, P1 attend une ressource en possession de P2, ... , et Pn-1 attend une ressource possédée par Pn lequel attend une ressource détenue par P0.
4- *Absence de réquisition*: Une ressource allouée à un processus Pi, ne peut être réquisitionnée avant sa libération naturelle (à la fin de l'exécution normale de Pi).

Remarque: En général, assurer la sécurité d'un système revient à identifier les problèmes indésirables pouvant l'affecter, et prendre les mesures appropriées pour qu'ils ne puissent survenir, ou à défaut, être en mesure de confiner ou minimiser les dégâts si jamais ceux-la se produisent. Ainsi, étant un problème indésirable, l'interblocage est considéré théoriquement comme faisant partie du domaine de la sécurité !

Exemple:
Soient deux processus P1 et P2 demandant l'allocation de deux types de ressources non partageables d'une installation, à savoir imprimantes et bandes magnétiques. En admettant que ces ressources existent chacune en exemplaire unique, le scénario d'exécution de P1 et P2 suivant conduit inévitablement à un interblocage (les quatres conditions précédentes sont simultanément vérifiées).

" **Processus1** "	" **Processus2** "
P(LP); demande de LP	**P**(MT) ; demande de MT
* **P**(MT); demande de MT, déjà occupé	(*) **P**(LP); demande de LP, déjà occupée

(*): à ce stade, les demandes ne peuvent être satisfaites; P1 bloque P2 qui à son tour bloque P1, c'est l'inévitable interblocage.

2.6.2 Privation 3.3.2

Un autre problème non moins important auquel peuvent être confrontés les programmeurs est celui que l'on désigne par: *famine* ou *privation* (starvation). Il désigne une situation dans laquelle un ou plusieurs processus concurrents, pour l'usage d'une ressource commune non partageable, se voit différer l'accès, pour une période pouvant être non acceptable, au profit d'autres processus (Cf. problème des lecteurs-rédacteurs). La privation agit donc à l' encontre de l'équité.

Pratiquement, la différence entre un interblocage et une privation peut être d'ordre temporel ou dynamique. Dans le premier, l'évolution de l'ensemble des processus bloqués est compromise tant qu'il n'y ait pas d'intervention extérieure; alors que dans le second, un ou plusieurs processus monopolisent le contrôle du processeur au détriment d'un ou plusieurs autres processus qui peuvent rester bloqués indéfiniment. Logiquement, le nombre de processus est fini, et un processus arrive au terme de son exécution au bout d'un temps fini (même dans trente ans!); donc, tôt ou tard le ou les processus en privation obtiendront le contrôle du processeur. La pénalité n'est donc que temporelle (pour une catégorie de processus), mais elle peut s'avérer grave dans un environnement qui ne peut supporter une telle contrainte (exemple: systèmes temps réels).

L'intervention précitée consiste en général à rompre le cycle de blocage mutuel par destruction délibérée d'un des processus impliqués. Deux stratégies peuvent être appliquées quant au choix du processus victime de la destruction.

- La première désigne comme victime le processus ayant accaparé le maximum de ressources; de cette manière, la libération de ces ressources permettrait l'évolution d'un nombre appréciable de processus bloqués. A noter que si le processus cible se trouve dans la phase terminale de son exécution, il subirait un préjudice certain. Cette stratégie qui a tendance à favoriser le degré de multiprogrammation s'avère intéressante dans les systèmes temps partagé.
- La deuxième stratégie considère comme cible de destruction le processus bloqué le plus jeûne possible. On espère ainsi pénaliser le moins possible la victime (le traitement à refaire est minimum). On peut considérer que le processus victime de la première stratégie est le moins jeûne si on estime que le nombre de ressources obtenues est une fonction croissante du temps d'exécution.

En général, il existe deux méthode de traitement de l'interblocage: Détection et guérison, et prévention. Dans le premier cas, tant que les ressources du système sont disponibles, elles sont allouées aux processus demandeurs. Lorsqu'un interblocage est détecté, on intervient pour le traiter. Dans le second cas, une ressource demandée n'est attribuée qu'après avoir vérifié que cela ne peut entraîner un interblocage. Cette vérification induit un overhead temporel mais permet d'avoir une stabilité au niveau du fonctionnement global. L'application de l'une ou l'autre méthode dépend étroitement des spécificités du système considéré.

Il est à noter que l'interblocage concerne aussi bien les processus utilisateurs que ceux du système d'exploitation lui-même. Si cela se produit, dans le premier cas, la destruction d'un processus, pour rompre l'état d'interblocage, ne pénalise que l'usager propriétaire du processus détruit et lui seulement. Dans le second cas, la destruction d'un processus du système devient beaucoup plus pénalisante et peut conduire à une réinitialisation du système tout entier (préjudice encouru par l'ensemble des usagers).

Différentes techniques de mise en oeuvre de l'expression de synchronisation ont été présentées, chacune d'elles permet, avec plus ou moins de souplesse et d'efficacité, aux processus de coopérer avec succès. L'usage de primitives ou de mécanismes de haut niveau intégrables dans des compilateurs serait d'un grand intérêt tant au niveau d'une expression de spécification claire et concise qu'au niveau de la validation de celle-ci.
Dans la suite seront présentés quelques outils de synchronisation de haut niveau ayant pour objectifs de pallier à certaines faiblesses rencontrées précédemment.

3.4 Exercices avec solutions

Exercice 1. Réalisez l'effet d'un sémaphore général (valeur initiale > 1) à l'aide d'un sémaphore privé (valeur initiale = 0) et d'un sémaphore de mutuelle exclusion (valeur initiale = 1).

Solution possible.
Ce sémaphore générale de valeur initiale supérieure à 1 doit être manipulé par les deux primitives: PP(semgen) et VV(semgen).

```
var semgen: integer init n;          %  n:  valeur initiale %
var mutex  : semaphore init 1,       %  Sémaphore de mutuelle exclusion %
    sempriv: semaphore init 0;       %  Sémaphore privé  %
PP(semgen):
begin
   P(mutex);
   semgen := semgen - 1;
   if  semgen ≥ 0 then V(sempriv);   (*)
   endif
   V(mutex);
   P(sempriv);
end;
```

(*): On émet un signal anticipé sur sempriv pour éviter le blocage d'un processus appelant PP avec une valeur de semgen ≥ 1; par contre, tout appel à PP avec semgen ≤ 0 donne lieu

au blocage de l'appelant (par P(sempriv)) en dehors de la section critique protégée par P(mutex) et V(mutex).

La primitive de déblocage associée à PP peut être:

```
VV(semgen):
 begin
    P(mutex);
    semgen := semgen +1;
    if semgen ≤ 0 then V(sempriv);
    endif
    V(mutex);
 end;
```

Exercice 2. Un sémaphore peut être passé comme paramètre lors d'un appel d'une procédure, ce passage doit-il se faire par valeur ou par adresse (référence, nom, variable)? Justifiez votre réponse.

Solution possible.
Un sémaphore n'est manipulable que par les primitives P et V, il doit donc être déclaré comme variable globale. Toute modification (autant de fois que c'est nécessaire) de sa valeur par l'une de ces deux primitives doit être perçue par l'autre. Toute Mise à jour doit donc être mémorisée. Ainsi, le passage doit se faire obligatoirement par adresse.

Exercice 3. Soient deux individus I1 et I2 se communiquant à travers un interphone ou tout autre support de communication utilisant les ondes radio(transmission half duplex). Ecrire la séquence algorithmique modélisant la communication entre i1 et i2.

Réponse: La communication étant half duplex donc alternée c'est-à-dire lorsque I1 parle I2 ne peut qu'écouter et réciproquement. Ainsi, les séquences de programme réalisant ce scénario de communication peuvent être les suivantes:

```
Var Atoi, Amoi : semaphore;
     Atoi := 0,  Amoi :=1;
Processus  I1                    Processus I2
      ¦                               ¦
Cycle                            Cycle
 P(Amoi)                          P(Atoi)
  < dialogue>                      < dialogue>
   V(Atoi)                          V(Amoi)
Fincycle                         Fincycle
```

Exercice 4. Discutez (en termes d'efficacité) l'implémentation des algorithmes P(sem) et V(sem) tels qu'ils ont été donnés au cours.

Exercice 5. Soient trois processus P1, P2, P3 se partageant quatre unités de ressource pouvant être allouées ou libérées une à une. Chacun des processus Pi ($1 \leq i \leq 3$) utilise au plus deux unités.
Question : Montrez que dans pareille situation l'interblocage ne peut survenir.

Solution possible

Un interblocage concerne une situation dans laquelle toutes les unités de ressource sont allouées tandis que un ou plusieurs processus se trouve en attente indéfinie pour l'acquisition de ressources supplémentaires. Autrement dit, chacun des processus concurrents monopolise l'usage d'un certain nombre d'unités et en demande d'autres déjà allouées qui ne peuvent être libérées. Toutefois, si les 04 unités sont allouées, alors un processus au moins a acquis 02 unités. Ce dernier terminerait son exécution au bout d'un temps fini et libèrerait les 02 unités. Cette libération permettrait à un autre processus en attente de poursuivre son exécution.

3.5 Conclusion

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 (dans un programme) des points de synchronisation.

- 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.

4. Régions Critiques

4.1 Introduction

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 leur 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.

Une région critique est donc une structure linguistique qui définit une séquence d'instructions à exécuter comme section critique dans laquelle sont manipulées des variables déclarées explicitement partageables.

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 regions critiques désignant ces variables.

4.2 Structure des régions critiques (syntaxe et sémantique):

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

Var V : ***shared*** T qui définit une variable V du type T;

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

Il est possible de considérer T comme une structure regroupant un ensemble de variables partagées distinctes (identiques relativement au partage) comme l'illustre la notation suivante:

```
Type direction = record          % Cf. exercice passage de véhicules sur un pont %
                   attente, passant, en_tete: integer
                 end
Var pont : shared record
                   gauchedroit, droitgauche : direction;
                 end
```

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

4.3 Région critique inconditionnelle

Region V ***do*** Sc

La sémantique exprime que la séquence d'instructions Sc dans laquelle est (ou sont) manipulée(s) la (ou les) variable(s) partagée(s) V, constitue une section critique (cf. § 2.4).

La séquence d'instructions Sc contient des variables déclarées partageables et éventuellement des variables locales au processus exécutant Sc.
Plusieurs régions critiques peuvent se rapporter à une même variable partagée; leurs exécutions s'excluent mutuellement. Autrement dit, si plusieurs processus tentent de les exécuter simultanément, un seul d'entre eux est autorisé à le faire (évolue normalement); les autres processus seront automatiquement bloqués. Lorsque le processus actif termine l'exécution de la séquence critique sc, un des processus en attente est systématiquement réactivé.

4.4 Région critique conditionnelle

4.4.1 ***Region*** V ***when*** cond ***do*** sc

Signifie que 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.

4.4.2 ***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 region 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 *Iv* 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'éxécution de Sc.

4.5 Exercices avec solutions

Exercice 1. Simuler un sémaphore à l'aide de régions critiques.

var sem: ***shared integer*** (sem ≥ 0);

P(sem): ***region*** sem ***do*** sem := sem - 1 ***await*** sem ≥ 0 ... (1) % incorrecte %

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 ≥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 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 2. 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;     %  ''                ''      reveillés                %
P(mutex);
if  not cond then begin
                        nbattente := 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(attente)
                                                 else   V(mutex);
                                                 P(attente);
                                              end;
                           nbattente := nbattente - 1;
                    end;
 sc;
if nbattente > 0  then  begin
                            nbreveil := 0;
                            V(attente);
                          end;
                  else V(mutex);
```

4.5 Conclusion

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 exécuteur. Ces dernières imposent donc que l'évaluation de la condition cond soit faite par le processus exécuteur. 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 restoration 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]

5. Moniteurs

5.1 Introduction

Le reproche que l'on peut adresser aux régions critiques concerne d'une part, le coût excessif relatif aux réveils systématiques des processus bloqués, en particulier dans des systèmes monoprocesseur, et d'autre part la dissémination des différentes instructions (au niveau des points de synchronisation) dans le code d'un processus. Ainsi, pour localiser tous les points de synchronisation afin d'examiner toutes les formes d'utilisation des ressources, il est impératif d'étudier la totalité des processus concurrents. Cette façon de faire est quelque peu contraignante et influe grandement sur le rendement du programmeur. Les moniteurs apportent une solution appréciable à ces problèmes en soulageant le programmeur de ces contraintes.

Ainsi, les moniteurs constituent la première approche qui tente une "séparation" entre la partie traitement et la partie contrôle correspondante d'un module fonctionnel dans un environnement concurrentiel. Autrement dit, un processus concurrent exécute normalement son algorithme ou sa partie traitement, et à la rencontre d'un point de synchronisation, il fait appel au moniteur pour obtenir l'accord du franchissement de ce point. le franchissement d'un point de synchronisation sera donc contrôlé au niveau du moniteur associé qui décidera de la poursuite ou du blocage de ce processus.

5.2 Définition

Un moniteur est une structure abstraite (module abstrait) composée d'un ensemble d'objets partagés et de procédures qui manipulent ces objets (données: variables, ...). Ces Objets (ou variables) partagés ne sont accessibles aux processus externes (appelants) que par l'intermédiaire des procédures du moniteur. Des données internes ou locales au moniteur peuvent exister. Elles peuvent être manipulées par des procédures internes au moniteur et invisibles de l'extérieur (du moniteur). Elles pourraient être utilisées pour faire changer d'état au moniteur.

5.3 Fonctions

L'exclusion mutuelle des procédures est assurée par le mécanisme d'exécution du moniteur. A chaque procédure peut être attachée une condition à laquelle est associée une file d'attente. Fonctionnellement, le moniteur encapsule dans une même entité les données et les procédures qui les utilisent. L'accès au moniteur est restreint à des appels de procédures externes (des procédures internes au moniteur peuvent exister et leur utilité peut être limitée à faire passer le moniteur dans un état bien déterminé). Tout moniteur doit comporter une section d'initialisation exécutée avant tout accès. Cette section initialisera les variables globales du moniteur.

En général, le moniteur est employé dans l'allocation de ressources ou servir d'outil de communication. Les procédures du moniteur ordonnent leurs actions au moyen de primitives de haut niveau: ***wait*** et ***signal***.

Plusieurs formes de moniteur ont été proposées pour réaliser la synchronisation de processus concurrents. Nous donnons ci-après la forme primaire due à Hoare [Hoa 74].

5.4 Primitives du moniteur

Wait : primitive de blocage.

La primitive Wait(cond), où *cond* représente la condition d'attente, met le processus appelant dans la file d'attente associée à cond, et libère l'exclusion mutuelle du moniteur pour permettre l'accès aux processus demandeurs externes.

Signal : Primitive de réveil.

La primitive ***signal***(cond) opère comme suit:

S'il n'existe aucun processus bloqué dans la file associée à la condition cond, alors le processus invocateur, soit P, continue en séquence. Dans le cas contraire, le processus P est temporairement bloqué et un processus en attente dans la file associée à cond est réactivé. Le processus invocateur du signal sera réveillé lorsque le moniteur devient libre; c'est-à-dire: le processus réveillé par signal vient de quitter le moniteur ou se bloque une nouvelle fois. De plus, le processus bloqué par signal devient prioritaire sur les processus externes qui tentent d'appeler le moniteur.

A remarquer qu'un événement signal n'est pas mémorisé contrairement aux sémaphores, autrement dit, un signal non attendu est purement perdu.

En général, il existe une fonction booléenne ***Empty***(condition) qui permet de tester la file d'attente associée à la condition; empty(condition) est vraie si la file d'attente est vide, fausse sinon.

5.5 Forme syntaxique du moniteur

```
nom_du_moniteur.Monitor
  % déclaration des variables %
Var    x:  Integer ,
          y: Boolean;
Condition c;

 % déclaration des procédures du moniteur %
procedure xx(yy);
begin
 ……..
end;
procedure zz(tt);
begin
……
end;
```

```
% initialisation  % la séquence d'instructions de la partie initialisation du moniteur %
begin
                    %  est exécutée avant tout appel de l'extérieur aux procédures du  %
                    % moniteur. Elle a pour but de mettre ce dernier dans un état      %
                    %  initial adéquat.                                                 %
end;
endmonitor
```

5.6 Difficulté du moniteur

L'inconvénient majeur de la spécification du moniteur précédente revient à l'emplacement de la primitive signal.

En effet, si cette primitive se trouve en un endroit quelconque dans le corps d'une procédure du moniteur, il se pose alors le problème du conflit entre le processus qui émet le signal de réveil (processus signalant) et le processus signalé (réveillé). Ces deux processus ne peuvent être actifs simultanément de peur de violer le caractère exclusif et essentiel de l'exécution dans le moniteur. Un des deux processus doit logiquement se bloquer au profit de l'autre. Ce blocage est imposé au processus signalant pour raison supplémentaire suivante:

Quand un processus est signalé, cela signifie que sa condition de franchissement de son point de synchronisation est devenue vraie (sa condition de blocage est fausse). S'il n'est pas immédiatement activé, le processus signalant, dans sa poursuite d'exécution, pourrait modifier cette condition. Si le processus signalé n'est activé qu'après sortie du moniteur du processus signalant, il serait dans l'obligation de tester sa condition de franchissement pour bien s'assurer qu'elle demeure encore vérifiée. Afin d'éviter ces tests fastidieux et répétitifs, le processus signalant sera donc bloqué après exécution de la primitive signal. Toutefois, le processus signalant sera réveillé à son tour avant tout autre nouvel accès au moniteur (voir, plus loin, la simulation du moniteur à l'aide de sémaphores).

Il faut noter que si la primitive signal est placée en fin de procédure, le problème de conflit ne se pose plus.

A noter également que ce problème demeure transparent au programmeur et incombe au concepteur de le faire prendre en charge par le mécanisme d'exécution du moniteur.

Kessels [Kes77] a proposé une variante du moniteur permettant de lever la contrainte précédemment citée.

Remarque: Si au niveau des sémaphores et régions critiques, un point de synchronisation peut se trouver à un endroit quelconque du code d'un processus et peut être associé à une séquence d'instructions (au besoin une instruction unique), dans le cas des moniteurs les points de synchronisation sont clairement localisés et s'appliquent à des séquences d'instructions plus importantes (procédures). La granularité de la synchronisation est donc

plus grande dans les moniteurs que dans les sémaphores et régions critiques; ce qui peut être gênant lorsque le temps d'exécution s'avère être un paramètre déterminant.

5.7 Variante de Kessels

La critique que l'on peut faire au mécanisme des moniteurs de Hoare concerne la primitive signal; en effet, le rôle de cette primitive consiste à réveiller, lorsque la condition de franchissement d'un point de synchronisation devient vraie, un processus bloqué en ce point. La spécification du fonctionnement de cette primitive a pour but d'assurer que cette condition de passage demeure inchangée entre l'exécution de signal et le réveil d'un processus en attente. Cette condition apparaît ainsi en deux endroits distincts du programme: avant le blocage et avant le réveil, ce qui peut être considéré comme un inconvénient. On observe également le manque de réveil en chaîne de plusieurs processus en attente de la même condition (Cf. problème des lecteurs-rédacteurs).
De plus, à la différence des sémaphores, un signal de réveil peut être purement perdu s'il n'est pas attendu, et un processus réveillé ignore l'état des variables globales. Il sait implicitement que la condition est devenue vraie.

Kessels [Kes77] proposa la variante suivante dans laquelle certains problèmes engendrés par la primitive signal sont évités par l'élimination de la primitive elle-même. On ne dispose donc que de l'unique primitive de blocage wait, et le réveil est à la charge du mécanisme d'exécution du moniteur qui le réalise de manière automatique. On aura donc:

Wait*(cond)*: où cond est maintenant une expression booléenne représentant de manière explicite la condition d'attente. Cond peut être composée de variables d'état du moniteur et éventuellement de constantes, elle peut être déclarée au début du moniteur. Cette primitive (wait(cond)) provoque le blocage du processus appelant P tantque la condition d'attente est vraie. Quand un processus quitte le moniteur ou se met en attente, les conditions associées aux primitives wait sont réévaluées. Si une des conditions examinées par le réévaluateur devient fausse, alors un des processus bloqué par cette condition est réveillé. Le réévaluateur est un processus du mécanisme d'exécution du moniteur qui se charge de la réévaluation des conditions de blocage.

Il est à remarquer que du fait du réveil automatique de processus subséquent aux réévaluations systématiques, il n'est pas aisé au programmeur d'intervenir pour mettre à profit la connaissance du changement d'état des variables globales afin de procéder à un réveil explicite. Cette situation peut faire de cette variante un outil complexe ou mal adapté à une certaine classe de problèmes facilement résolubles à l'aide du moniteur classique (voire problème de l'allocateur de ressources dans la suite).

4.8 Exercices corrigés

Exercice 1. Simulez un sémaphore à l'aide de:

a) Moniteur classique,

b) Variante de Kessels.

Solution possible

a) Moniteur classique (Hoare)

```
Semaphore.Monitor
begin
   Var  nbreqP, nbopPex, nbopV: integer
   cond : condition;
              % nbreqP = nombre de demandes d'exécution de la procédure P %
              % nbopPex = nombre d'opérations P effectuées                 %
              % nbopV   = nombre d'opération V effectuées                  %
Procedure P
  begin
    nbreqP := nbreqP + 1;
    si nbreqP > nbopV alors  wait(cond);
    fsi
    nbopPex := nbopPex + 1;
  end
Procedure V
  begin
    nbopV := nbopV + 1;
    si nbreqP > nbopPex alors signal(cond);
    fsi
  end
% Initialisation %
  begin
    nbreqP := 0;
    nbopPex  := 0
    nbopV  := < valeur initiale du sémaphore >;
  end
endmonitor.
```

La solution évidente est celle transcrite directement de l'algorithme des primitives P et V.

```
Sem.Monitor
begin
  var Comptsem : integer;
      cond     : condition;
Procedure P
```

```
 begin
   comptsem := comptsem - 1;
   if comptsem < 0 then wait (cond)
   fi
 end
Procedure V
 begin
   comptsem := comptsem + 1;
   if comptsem ≤ 0 then signal(cond);
   fi
 end
 comptsem := < valeur initiale du sémaphore >;
endmonitor.
```

b) Variante de Kessels

La version possible à l'aide des moniteurs de Kessel peut s'écrire comme suit:

```
Sem.Monitor
begin
  var comptsem : integer ;
     condition  cond: comptsem ≤ 0;       % Condition explicite d'attente %
procedure P
  begin
    wait(cond);
    comptsem := comptsem - 1;
  end
procedure V
  begin
    comptsem := comptsem + 1;
  end
 comptsem := < valeur initiale >  % initialisation %
end monitor.
```

-Version utilisant le nombre d'opération V et P

```
Semaphore.Monitor
begin
  Var nbreqP, nbopV: integer
  condition cond:  nbreqP > nbopV;

  Procedure P
    begin
      nbreqP := nbreqP + 1;
```

```
    Wait(cond);
  end

 Procedure V
  begin
    nbopV := nbopV + 1;
  end

% Initialisation  %
  begin
    nbreqP  := 0;
    nbopV   := < valeur initiale du sémaphore >;
  end
endmonitor.
```

Remarque:
L'utilisation des procédures du moniteur sémaphore se fait par appel comme suit:

call semaphore.P
call semaphore.V

Exercice 2. Simulez un moniteur à l'aide de sémaphores.

Pour implémenter un moniteur en utilisant des sémaphores, il est nécessaire de distinguer deux cas liés aux problèmes soulevés par la primitive signal.

cas simple: La primitive signal figure comme dernière "instruction" d'une procédure du moniteur.

L'exclusion mutuelle lors des exécutions des procédures du moniteur est obtenue en entourant chacune d'elles par un prologue et un épilogue. ces deux derniers, représentés respectivement par les opérations P et V sur un sémaphore d'exclusion mutuelle associé au moniteur, sont établis par le compilateur au moment de la compilation des programmes.
Toute procédure PROCi du moniteur se présentera comme suit (avec: sémaphore mutex initialisé a 1):

```
P(mutex)  ------> prologue;
  .
  .
corps de PROCi
  .
  .
V(mutex)  ------> épilogue;
```

Quant au blocage des processus devant la primitive wait et leurs réveils après exécution de la primitive signal correspondante, leur réalisation peut se faire de la manière suivante:

On associe à chaque condition (cond) de franchissement d'un point de synchronisation, un sémaphore privé semprive de valeur initiale 0, et un compteur compt qui comptabilise le nombre de processus bloqués dans la file associée à cond. L'attente de cond peut s'exprimer alors:

"***wait*(cond)"**

```
compt := compt + 1;   %incrementation du nombre de processus bloqués %
V(mutex);             % libération de l'exclusion mutuelle     %
P(semprive);          % puis blocage du processus  en cours %
compt := compt - 1;   % mise à jour de compt  après  réveil  %
```

Le réveil d'un processus par signal(cond) s'exprime également comme suit:

"***signal*(cond)"**

```
if compt > 0 then     %    s' il existe au moins un processus bloqué    %
  V(semprive)         %    le réveiller et lui attribuer le contrôle    %
else                  %    du  processeur. Noter  que l'exclusion       %
  V(mutex);           %  mutuelle n'est pas libérée, ce qui confère     %
                      %  une  priorité au processus réveillé devant     %
                      %  ceux qui attendent l'entrée dans le moniteur   %
```

Cas moins simple: La primitive signal n'est pas en fin de procédure.

Pour les raisons citées précédemment (conflit d'exclusivité entre processus signalant P et processus signalé Q) où on impose que la primitive signal devienne bloquante, le mécanisme d'exécution du moniteur doit attribuer une forte priorité à P de sorte que lorsque Q se bloque de nouveau ou quitte le moniteur, P doit reprendre l'accès au moniteur (évitement de famine de P).

Pour ce faire, on utilise un sémaphore semprioritaire initialisé à 0 et associé au moniteur pour bloquer tout processus signalant P. On aura également besoin d'un compteur nbprocprio qui dénombre les processus du type P. Toute procédure du moniteur doit se conformer au schéma suivant:

```
P(mutex);     %  prologue
  ……
 PROCi
  ……
if  nbprocprio > 0  then
   V(semprioritaire);
```

else % épilogue
V(mutex);
endif

La primitive *wait*(cond) et *signal*(cond) peuvent s'écrire comme suit:

"***wait*(cond)**":
compt := compt + 1; % incrémentation du nombre de processus bloqués par wait %
if nbprocprio > 0 **then** % S'il existe des processus signalant bloqués en réveillé un %
V(semprioritaire);
else
V(mutex)
P(semprive);
compt := compt - 1;

"***signal*(cond)**":
nbprocprio := nbprocprio + 1; % Incrémentation du nombre de processus signalants %
if compt > 0 ***then*** % si nombre de processus bloqués par wait est > 0 %
begin
V(semprive) % Réveil d'un processus bloqué par wait %
P(semprioritaire) % blocage d'un processus signalant %
end
endif
nbprocprio := nbprocprio - 1; % un processus signalant sera reveillé %

6. Mecanismes de Synchronisation de Haut Niveau

6.1 Modules de contrôle

Le principe du *module de contrôle* (mdc) [Rob77], basé sur la séparation du contrôle et du traitement, consiste à regrouper: d'une part, dans un module (module de traitement) un ensemble de procédures et d'autre part, dans un module de contrôle les règles de synchronisation correspondantes. Bien entendu, la possibilité de rattacher un module de traitement dès sa création au module de traitement associé est disponible.

Le module de contrôle qui ne connaît que les noms des procédures à synchroniser (interface) sera activé à chaque changement d'état des composants du module de traitement. Ces changements d'état sont associés aux événements ayant trait aux requêtes, autorisations, et terminaisons des exécutions des procédures formant le module de traitement (mdt).

Le langage de spécification du contrôle des accès aux objets partagés associe à chaque procédure manipulée par le module de contrôle, une condition d'executabilité et une file d'attente dans laquelle sera bloqué le processus appelant ne satisfaisant pas la condition. Pour chaque procédure, on dispose de cinq compteurs qui mémorisent l'historique des accès au module. Initialement ces compteurs sont nuls et croissent de façon monotone; la file d'attente étant vide. Il est possible de modifier la restriction des valeurs initiales des compteurs au moyen d'une instruction appropriée.

Les compteurs qui décrivent les règles de synchronisation sont les suivants:

req(p) : nombre total de requêtes d'exécution d'une procédure p depuis l'initialisation du module de contrôle.
aut(p) : nombre total d'autorisations données pour l'exécution de p depuis l'initialisation.
term(p): nombre total de terminaisons d'exécution de p depuis l'initialisation.
act(p) : nombre d'exécutions en cours de p à un instant donnée (p étant réentrante):

act(p) = aut(p) - term(p)

att(p) : nombre de requêtes enregistrées mais non autorisées

att(p) = req(p) - aut(p)

Au niveau du contrôle des accès à un objet partagé, ces différents compteurs sont associés à trois événements caractéristiques suivants:
requête(p)i : ième requête de la procédure P;
autorisation(p)i : ième invocation de P déclenche son exécution;
terminaison(p)i : ième invocation de P termine son exécution;

Puisque les événements sont sérialisés comme suit:
requête(p)i --> autorisation(p)i --> terminaison(p)i et ce, pour toute procédure P et l'instant i, on en déduit: $term(p) \leq aut(p) \leq req(p)$.

Le mdc (module de contrôle) doit enregistrer les demandes (requêtes) et les terminaisons d'exécutions selon leurs arrivées. Son pouvoir de décision est exercé seulement aux autorisations. On doit donc donner pour chaque procédure (ou ensemble de procédures) sous le contrôle du mdc, un ensemble de conditions nécessaires pour que chacune de ces requêtes d'exécutions soit autorisée. Une condition est une expression booléenne formée de compteurs et de constantes.
Le réveil des processus bloqués s'effectue par le mdc suite à des événements enregistrés et ayant provoqués un changement des conditions d'autorisations d'exécution (un peu à la manière des régions critiques).

Exemple : Problème du producteur-consommateur.

Spécification des règles de synchronisation:
1. Le consommateur ne peut consommer (s'exécuter) plus que ce qu'a produit le producteur d'où: aut(consommer) ≤ term(produire).
2. Le nombre de productions permises ne peut excéder de n le nombre de consommations i.e. : aut(produire) ≤ term(consommer) + n.

De 1., on en déduit la condition d'autorisation de consommer:
term(produire) - aut(consommer) > 0.
De 2., il en résulte la condition pour produire: aut(produire) < term(consommer) + n.
Les mdc n'assurent pas une exclusion mutuelle à produire et consommer, contrairement aux moniteurs, il va falloir l'imposer en le spécifiant par: act(produire) + act(consommer) = 0. Le module_de_contrôle du couple producteur-consommateur peut donc s'écrire:

```
prodcons: module_de_controle
  begin
    produire, consommer : procedure;
    faprod, facons      : queue;
    condition(produire) : aut(produire)-term(consommer)<n
                  and act(produire)+act(consommer)=0; (*)
    condition(consommer): term(produire)-aut(consommer)>0
                  and act(produire)+act(consommer)=0; (*)
  end
end module_de_controle.
```

Pour exprimer la manipulation par une procédure P d'un objet partagé tout en garantissant la cohérence et l'intégrité de ce dernier, on doit écrire l'invariant suivant: act(P) ≤ 1. De même, pour exprimer la condition d'exécution en section critique de chacune des deux procédures P et Q, on doit écrire: act(P) + act(Q) = 0 (Cf. (*) précédent)

6.1.1 Avantages des mdc

L'avantage le plus important est sans doute la séparation entre les règles qui expriment la synchronisation (mdc) et les actions du traitement sur lesquelles porte cette synchronisation (mdt). Cette séparation va permettre une flexibilité dans la maintenance des mdt c'est-à-dire permettre des modifications des composants d'un mdt sans incidence sur le mdc associé. On observera également une certaine aisance au niveau de la programmation puisque les algorithmes de traitement pourront être écrits sans tenir compte du contrôle qui lui sera remis à un moment ultérieur.

Un autre avantage de cette séparation est la possibilité d'établir des preuves et des invariants. Autrement dit, déterminer qu'une spécification du contrôle est exempte des problèmes d'interblocage ou de privation.

6.1.2 Inconvénients

L'inconvénient que l'on peut adresser au mdc (outre, sa difficulté d'implémentation) est d'être restrictif quant à la classe de problèmes où il peut répondre aisément (en comparaison par exemple avec l'outil sémaphore).

En effet, si les règles de synchronisation ne s'exprime pas facilement en terme d'usage d'un objet partagé servant de support de communication d'information, il serait difficile de résoudre le problème sans avoir à changer de niveau d'observation. Ce dernier peut contraindre à user de moyens artificiels allant jusqu'à dénaturer ce problème. C'est notamment le cas du problème de l'allocateur de ressources (voir plus loin) où le blocage et le réveil de processus se fait à travers la mise en place de procédures artificielles à corps vide. La programmation d'un modèle de coordination de processus (par exemple le problème des rendez-vous en ADA) nécessiterait inévitablement une certaine gymnastique.

6.2 Expressions de chemin

Le concept fondamental du mécanisme des expressions de chemin (path expressions [Hab74]) consiste à décrire les règles de synchronisation qui expriment un ordonnancement à respecter par des processus concurrents, lors des accès à des objets communs (partageables), au moyen d'expressions. L'intérêt d'une telle spécification réside (comme les modules de contrôles) dans la séparation de l'expression du contrôle d'accès à l'objet partagé, de la manipulation de cet objet dans un algorithme de traitement. Les avantages de cette séparation ne sont plus à démontrer, citons en un en particulier, celui de la flexibilité des modifications des procédures de traitement sans qu'il y ait incidence sur le contrôleur qui assure la synchronisation.

Une expression de chemin contient une spécification qui décrit exactement l'organisation de la synchronisation. Autrement dit, la synchronisation est spécifiée directement au niveau des procédures (interface du contrôleur) et elle décrit comment une procédure (considérée comme unité vis-à-vis du contrôleur) sera exécutée en tenant compte des autres, et indépendamment des processus invocateurs.

Le langage des expressions de chemins fournit des opérateurs de contrôle qui reçoivent comme arguments les noms des procédures qui accèdent à l'objet à contrôler. Ces opérateurs sont les suivants:
Soient P et Q les procédures dont on désire synchroniser les accès à un objet commun.

1. *opérateur de séquence*: **;** (point virgule)

P ; Q signifie: les procédures P et Q doivent s'exécuter de manière séquentielle et dans l'ordre indiqué c'est-à-dire P puis Q. Si un processus appelle Q il sera mis en attente jusqu'à la fin de l'exécution de P.

2. *opérateur de sélection*: , (virgule) P , Q signifie: exécution des procédures P et Q dans un ordre quelconque en *exclusion mutuelle*. Un processus appelant l'une de ces procédures (P ou Q) l'exécute sans attendre; tout autre appel subséquent met l'appelant en attente jusqu'à terminaison du premier.

Il est possible d'obtenir des expressions plus complexes par composition de séquence, de sélection et de groupement à l'aide de parenthèses, par exemples: P, (Q ; R) , S exprime que l'on peut exécuter soit P, soit Q suivie de R, soit S

3. *opérateur d'exécutions multiples*: { } (accolades): { P } indique que le nombre d'exécutions de P à un instant donné est quelconque (absence d'exclusion mutuelle).

4. *itération*: ***Path*** expr ***end*** signifie: une fois l'évaluation de l'expression expr est terminée, elle peut être réévaluée de nouveau.

Par exemple: L'expression de chemin suivante:
Path P ; (***Path*** Q ; R ***end***) ; S ***end*** permet les séquences d'exécution suivantes:
Soit P **;** S et zéro fois l'exécution du chemin interne,
Soit P ; Q ; R ; S
Ou bien P ; Q ; R ; Q ; R ; S , etc ...

6.2.1 Puissance et limites des EdC

Fondés sur la séparation du contrôle et du traitement, Comme pour les MdC, les EdC offrent un outil puissant pour l'expression de la spécification des règles de synchronisation réalisant l'intégrité lors des accès aux objets partagés. Contrairement aux MdC, les règles décrivant les priorités d'accès s'expriment difficilement par les EdC. Cette difficulté tient au fait que les opérateurs du langage des EdC (excepté la simultanéité: { }) s'adresse uniquement au séquencement des exécutions des procédures.
Pour exprimer la priorité, il faut procéder par ajout de procédures secondaires sur lesquelles portera le séquencement qui réalise la priorité désirée sur les procédures primaires. Cette façon de faire qui contourne le problème en spécifiant une mise en oeuvre au lieu du problème lui-même s'avère contraignante et parfois complexe. Cette complexité

peut être une source d'erreurs dans la détermination d'invariants pour la vérification et la validation des règles de synchronisation. A titre d'exemple voir le problème des lecteurs-rédacteurs avec priorité des lecteurs.

Des extensions des EdC ont été proposées en particulier:

- Les chemins conditionnels qui permettent de sélectionner un élément de chemin (une procédure ou bien un sous ensemble de procédures) en fonction des valeurs courantes du module.

- Les conditions bloquantes permettent à un processus de se bloquer soit avant, soit après une exécution de procédure en attendant qu'une certaine condition devienne vraie.

- Chemins numériques [flo 76] qui permettent d'exprimer des relations sur le nombre d'exécutions des procédures. par exemple: le chemin ***Path*** (produire - consommer)n ***end*** impose que la différence entre les nombres d'exécutions de produire et consommer soit compris entre 0 et n, soit: 0 ≤ nombre_de_produire - nombre_de_consommer ≤ n.

Exercice 1. Simulez les primitives P et V de manipulation de sémaphores à l'aide des expressions de chemin.

Solution

```
Sem.EdC
begin
  type element = record  begin wait  : procedure; {corps vide}
                               resume: procedure; {corps vide}
                               path resume; wait end;
                 end
 var  compsem  : integer  init  val;      % valeur initiale %
      process  : identité_processus;
      strucproc: array [process] of element;
      fileatt  : queue init empty;

{déclaration des procédures enqueue (i, fileatt) et dequeue(i,fileatt) respectivement pour
enfiler un processus i en attente dans la file fileatt (gérée en fifo), et extraire un processus
de fileatt avec identité rendue dans j }

external procedure P(i:process)
begin
  PP(i);
  strucproc[i].wait;
end
```

```
procedure PP(i)
begin
  compsem := compsem - 1;
  if compsem < 0 then   enqueue(i, fileatt)
  else strucproc[i].resume
  endif
end

external procedure V
var j : process_name;
begin
  compsem := compsem + 1;
  if compsem ≤ 0 then begin
                       dequeue(j, fileatt);
                       strucproc[j].resume;
                      end
  endif
end
end sem.

path PP, V end
```

La simulation précédente est fidèle aux algorithmes des primitives P et V donnés précédemment. Toutefois, une autre simulation plus simple (cas particulier), basée sur la sémantique des expressions de chemin est comme suit:

Chaque variable du type sémaphore doit avoir son propre chemin dont la description peut être la suivante:

```
type semaphore = begin
                   procedure V : (corps vide);
                   procedure P : (corps vide);
                   path { V ; P } end
                 end
```

La valeur du sémaphore ne peut être modifiée que par l'exécution de P et V (respect du concept sémaphore). Comme l'indique le chemin, toute exécution de P entraîne un blocage de l'appelant dont le réveil est subséquent à une exécution de V. Ceci implique que la valeur initiale du sémaphore est nulle (sémaphore privé). Pour obtenir un sémaphore de mutuelle exclusion, il suffit d'inverser l'ordre de P et V dans le chemin précédent: ***path*** {P;V} ***end***. La valeur du sémaphore étant initialisée à 1.

6.3 Difficultés des Mécanismes de Synchronisation

A travers la chronologie des quelques différents outils de synchronisation qui ont été présentés, est montrée la tendance d'évolution des mécanismes d'un niveau bas (tels les sémaphores) vers un haut niveau (mdc ...). A noter qu'il ne faut pas perdre de vue le niveau implicite fondamental sur lequel repose tout autre niveau supérieur, qu'est celui de l'arbitrage (matériel) des accès aux cellules mémoire. Cette tendance vise particulièrement les aspects suivants:

- Clarté dans la localisation des points de synchronisation;
- Moindre contribution du programmeur à introduire des fautes par maladresse ou par mégarde en manipulant un outil de haut niveau.
- Souplesse dans la maintenance des modules logiciels. Ce critère est à la base de la séparation entre la partie fonction ou traitement d'un module et sa partie contrôle ou synchronisation. Autrement dit, une mise à jour des algorithmes de traitement peut se faire sans incidence sur la partie contrôle.
- Capacité accrue d'établissement d'invariants servant notamment à valider le bon déroulement des activités; en l'occurrence vérifier certaine propriétés telles que: absence d'interblocage, assurance de l'exclusion mutuelle, etc ...

  La validation de ces propriétés peut être confiée à des automatismes.
- Intégration de ces mécanismes dans des langages de haut niveau, ce qui banaliserait leur emploi, et ferait faire certaines vérifications au stade de la compilation.

A noter que l'évolution du bas niveau vers les niveaux supérieurs faciliterait d'avantage la tache du programmeur au détriment d'une complexité d'implémentation concédée au concepteur. Il faut également remarquer que la notion de temps et de sûreté de fonctionnement dans les mécanismes de synchronisation ne sont pas abordés de manière explicite. Ainsi que se passerait-il si un processus devient subitement défaillant à l'intérieur d'une section critique; ou qu'il doit en sortir (normalement) tout en respectant une contrainte temporelle. Ces problèmes vont être présentés sommairement dans la suite.

6.3.1 Tolérance aux fautes

Il est admis qu'un processus qui entre dans sa section critique doit en ressortir (la libérer) au bout d'un temps fini. Cette hypothèse importante suppose implicitement que ce processus est correct et exempt de fautes; ce qui peut conduire à considérer l'alternative suivante:

- Soit qu'une fiabilité "démesurée" est accordée au programmeur quant à la justesse et la correction de ses algorithmes et de l'écriture du code correspondant au processus, ce qui est évidemment réfutable.
- Soit que le problème d'anomalie ou de défaillance d'un processus dans une section critique est laissé à lors de l'implémentation, et donc résolu au coup par coup.

En tout état de cause ce problème de défaillance à l'intérieur d'une section critique n'a pas été pris en compte (ou très peu relaté) immédiatement lors de la conception des outils de synchronisation (les auteurs n'en parlent pas ou en font état de manière très brève en particulier dans un environnement centralisé).

Le problème de sûreté de fonctionnement d'un processus (en concurrence pour la manipulation d'un objet commun) en particulier dans une section critique est assez compliqué. Cette complexité relève de deux sources possibles de fautes à savoir: le programmeur et le matériel.

En effet, d'aucuns ne peut prétendre à une écriture de logiciels sûrement correcte malgré les récents développements (non encore satisfaisants) des outils de spécification et de validation formels. Faut-il encore ajouter à cela l'aspect matériel d'un système (tout système informatique est composé d'un système abstrait et d'un système physique).

Partant du principe, communément admis, qu'on ne peut assurer qu'un système est totalement correcte (zéro défaut); car il existe toujours des fautes résiduelles d'origine matérielle ou logicielle et des fautes matérielles d'origine environnementale, notamment à caractère transitoire ou intermittent, il est d'usage de pouvoir y survivre en les tolérant.

Ainsi, puisqu'on ne peut les éliminer, il faut pouvoir y faire face lors de leurs manifestations sous forme d'erreurs pouvant conduire à des situations préjudiciables et parfois dangereuses. C'est pourquoi, l'intégration de ces problèmes au sein des mécanismes de synchronisation est souhaitable et même indispensable dans la programmation des systèmes à très haut degré de sécurité (où une défaillance peut conduire à une catastrophe).

Nous n'allons pas présenter les techniques de tolérance aux fautes car tel n'est pas notre objectif, mais en parler sommairement pour bien soulever le problème.

En général, avant d'autoriser un processus d'entrer dans une section critique, on identifie les objets communs qui y seront manipulés et on procède à une sauvegarde. En cas de défaillance du processus, le système le contraint à libérer la section critique, et restaure les objets communs à leurs valeurs initiales (avant la manipulation). Autrement dit, on doit assurer que les actions effectuées sur les objets partagés, dans une section critique, respectent le principe du *tout ou rien*; c'est à dire que la mise à jour de ces objets est correcte ou n'ait jamais eu lieu. Cette notion de correction est très délicate. Elle est difficile à obtenir de manière absolue car ceci implique un ajout de tests de validation sémantique qui pourraient augmenter considérablement le code de la section critique. Cette augmentation risque de dénaturer le concept de section critique qui stipule que cette dernière soit la plus brève possible afin de permettre d'avantage de simultanéité dans les exécutions.

Afin de pouvoir fournir au programmeur des outils de synchronisation capables de résister aux défaillances, quelque soient leurs origines, les concepteurs de systèmes doivent en tenir compte lors de l'implémentation. Ce problème s'avère d'une importance particulière à mesure que le développement des systèmes répartis (réseaux) prenne d'avantage d'ampleur.

Le concept de transaction en est l'exemple; il permet d'assurer une résistance aux défaillances aussi bien dans un environnement centralisé que réparti. Il soulage le programmeur de ces problèmes d'erreurs de manière transparente, et s'applique aux systèmes à caractère transactionnel (systèmes de gestions de bases de données, systèmes transactionnels, ...).

6.3.2 Contraintes temporelles

Les mécanismes de synchronisation présentés sont applicables notamment dans des systèmes à usage général, en ce sens, des difficultés peuvent surgir quant à leur adoption dans des systèmes spécialisés.

En effet, certains impératifs, en particuliers les contraintes temporelles, n'y sont pas spécifiquement considérées. Il peut s'avérer que ces mécanismes ne puissent répondre de manière favorable dans les systèmes à temps contraint (temps réel).

Cette inadéquation est d'autant plus importante à mesure que l'on s'éloigne du niveau bas vers les niveaux supérieurs. Ainsi, le mécanisme d'exécution d'un contrôleur de synchronisation dans les modules de contrôle exige d'avantage de temps que les primitives de manipulation des sémaphores (qui elles même s'avèrent parfois temporellement exigeantes). Bien entendu, le principe des priorités accordées aux processus concurrents allège quelque peu cette contrainte temporelle sans toutefois donner entière satisfaction dans un environnement temporellement stricte. Dans pareils cas, il est donc évident de bien choisir un outil de synchronisation le moins temporellement pénalisant et au besoin d'en concevoir les mieux adaptés (réduire au maximum le temps d'attente d'entrer en section critique). Il serait donc souhaitable de concilier à la fois la souplesse, temporellement pénalisante des mécanismes de haut niveau, et la rigidité temporellement efficace des mécanismes de bas niveau. Bien entendu, il est toujours possible d'utiliser les outils matériels peu flexibles, tels que: les instructions TAS, Exchange, etc.

En définitive, et dans l'état actuel de l'art, des mécanismes satisfaisants et bien adaptés aux environnements temps réel restent encore à mettre à jour.

6.3.3 Problèmes d'interblocage

Comme cité précédemment, la situation d'interblocage considéré comme un problème de « sécurité » provoque souvent un préjudice, plus ou moins notable, fonction de l'importance accordée à l'évolution normale des processus mis en cause. En ce sens, lorsque deux ou plusieurs processus d'une même application se trouvent en situation d'interblocage, leur traitement implique inévitablement la destruction de l'un d'entre eux. Cette destruction peut conduire à l'annulation de toute l'application si le processus détruit s'avère jouer un rôle essentiel dans le fonctionnement normale de l'application.

Les conséquences d'annulation d'une application (dommages engendrés) dépendent du caractère critique que revêt l'accomplissement de la mission assuré par cette application.

- Ces dommages peuvent se limiter à un simple préjudice au niveau temporel, c'est-à-dire: l'application sera de nouveau lancée et les résultats attendus seront produits avec un retard

acceptable. Il faut noter que ce retard peut ne pas être acceptable, en particulier dans des applications dites *temps réel*, où le facteur temps est aussi important que les résultats corrects fournis. Autrement dit, les résultats attendus sont acceptables s'ils correctes et délivrés dans les délais impartis.

- Les dommages subits peuvent être d'une extrême importance pouvant aller d'une perte d'argent (ou manque à gagner) à une hypothétique perte de vie humaine.

A titre d'exemple, des processus interbloqués d'une application bancaire peuvent conduirent à l'incapacité d'assurer les transactions qui y sont impliquées (préjudice financier). Par contre, si l'interblocage affecte des processus d'une application ou d'un système de contrôle et suivi de « processus » industriels ou systèmes embarqués, les conséquences peuvent être catastrophiques.

Si les processus interbloqués appartiennent au système de base, il se peut que la solution ne puisse provenir que d'une réinitialisation de ce dernier, avec toutes les implications que cela peut engendrer au niveau global de toutes les exécutions en cours! Ainsi, l'importance de ce problème (interblocage) est donc liée à l'importance du déroulement normal des applications et du système qui les contrôle.

L'interblocage constitue toujours un problème délicat, aussi son traitement dépend donc des spécificités de l'application. Si cette dernière ne peut accepter qu'un processus actif soit détruit à cause d'interblocage, il faut alors faire en sorte que l'interblocage ne puisse survenir, d'où la mise en oeuvre d'algorithmes assurant son *évitement* par *prévention*.

Si les algorithmes d'évitement et de prévention empêchent certes, l'interblocage de se produire, ils sont souvent coûteux en temps. Ce coût temporel est supporté par l'ensemble des processus: ceux du système et des applications qui s'y exécutent.

Lorsqu'on désire seulement pénaliser les applications dont les processus ont provoqué l'interblocage, sans incidence sur d'autres applications, il peut être préférable de laisser l'interblocage se produire puis de le traiter en conséquence, c'est la politique dite de *traitement* et *guérison*.

7. Tests des connaissances

7.1 Examen 1

Exercice 1

Considérons le problème producteur/consommateur, vu en cours et en td.
1) Adaptez la solution suivante au cas d'un seul producteur, n consommateurs et un tampon à n cases de même taille. Chaque message produit par le producteur est déposé dans toutes les cases du tampon en commençant par la première. Le consommateur i récupère (consomme) les messages déposés dans la case i.
Variables :

Mutex: Sémaphore binaire, Vide: Sémaphore général init n, Plein: Sémaphore privé;
tampon : tableau[1..n] de caractères;

Producteur ( )
```
int ip =0 ;
cycle
  debut
    P(Vide) ;
    P(Mutex) ;
    Tampon[ip]=msg;
    V(Mutex) ;
     ip++ ;
    V(Plein) ;
  Fin
Fincycle
```

Consommateur( )
```
int ic =0 ;
          cycle
   debut
      P(Plein) ;
      P(Mutex) ;
       m = Tampon[ic];
       V(Mutex) ;
        ic++ ;
       V(Vide) ;
    fin
     Fincycle
```

2) en utilisant les moniteurs classique (Hoare), donnez la solution à la question 1.

Exercice 2

Résoudre le problème des lecteurs/rédacteurs avec l'équité (priorité égale) en utilisant :

a) Moniteur de Hoare
b) Modules de contrôle

Exercice 3

On considère deux producteurs P1 et P2 qui produisent des messages et les déposent dans deux tampons T1 et T2 respectivement (T*i* pour P*i*, *i*=1,2). Deux processus consommateurs C1 et C2 consomment les messages : C1 ceux de T1, C2 ceux de T2 ; avec la contrainte que lorsqu'un processus C*i* (*i*=1,2) consomme un message, il attendra que l'autre processus C*j* (*j*=3-*i*) ait consommé un message lui aussi pour continuer à consommer un autre message (Rendez-vous entre C1 et C2 après chaque consommation).
Synchroniser ces processus en utilisant :

a) Les sémaphores.
b) Les régions critiques
c) Les moniteurs de Kessels

7.2 Examen 2

Exercice 1

Soit le problème des philosophes, supposons que certains philosophes sont gauchers (ils mangent avec la main gauche) et d'autres sont droitiers (ils mangent avec la main droite). Supposons également qu'il y a au moins un gaucher et un droitier à la table. L'interblocage peut-il se produire? Justifier votre réponse.

Exercice 2

Pour réaliser leurs projets PFE, deux groupes d'étudiants (groupe licence et groupe master) utilisent la même salle TP1 dans laquelle se trouve N ordinateurs. A tout moment Il ne doit y avoir que des étudiants du même groupe, soit groupe licence (GL) ou groupe master (GM), dans la salle TP1. Aussi on ne peut avoir plus de N

étudiants travaillant en même temps dans la salle TP1. Le nombre d'étudiants GL est supérieur à N mais celui des étudiants GM est inférieur à N. Corriger la solution suivante qui a été proposée par un étudiant.

GL-GM.Moniteur

```
Var Entrer-GL, Sortir-GL, Entrer-GM, sortir-GM : procedure ;
    nbGM, nbGL : entier ;
    GM, attent : condition
```

Procedure Entrer-GL

```
Debut
Si nbGM ≠ 0 alors
  Wait(GL)
Finsi
Si nbGL < N alors
   nbGL++
sinon
   natt++
 Finsi
Fin
```

Prcedure Sortit-GL

```
Debut
 nbGL - -
 si natt ≠ 0 alors
   signal (attent)
 sinon
   si nbGL =0 alors
      signal(GM)
    finsi
  finsi
Fin
```

Procedure Entrer-GM

```
Debut
  Si nbGL ≠ 0 alors
   Wait(cond)
  Finsi
   nbGM++
Fin
```

Procedure Sortir-GM

```
Debut
   nbGLM - -
 Si nbGM = 0 alors
   Wait(cond)
  Finsi
Fin
```

7.3 Examen 3

Exercice 1

Soit P0 et P1 deux processus parallèles se partageant deux ressources R1 et R2. Les algorithmes de ces deux processus sont écrits comme suit :

```
var s1,s2 : sémaphore init 1,1 ;
Processus P0              Processus P1
Début                     Début
(a0) P(s1)                (a1)  P(s2)
     utiliser R1                utiliser R2
     P(s2)                      P(s1)
     utiliser R1 et R2          utiliser R1 et R2
     V(s1)                      V(s2)
     V(s2)                      V(s1)
    Aller_à a0                 Aller_à a1
Fin                       Fin
```

1) à quelle situation anormale peut conduire l'exécution de ces deux processus ? Justifier votre réponse
2) donner une solution à ce problème.

Exercice 2

Un parking de véhicules peut accueillir un nombre limité de voitures. Cette limite est représentée par le nombre N de places disponibles. Un véhicule qui arrive à l'entrée du parking attend s'il n'y a aucune place disponible sinon il y entre. Tout véhicule quitte le parking dès qu'il le désire.
On considère qu'il y a deux catégories de véhicules : les abonnés et les non abonnés. Il n'y a pas d'exclusion mutuelle entre abonnés et non abonnés , par contre les abonnés ont la priorité pour l'acquisition des places.
On peut assimiler les véhicules à des processus parallèles et les places à des ressources partagées.
Compléter les algorithmes des processus « abonnés » et « non abonnés » donnés cidessous.

Parking.Moniteur

```
Var Entrer-Abonné, Sortir-Voiture, Entrer-Non-Abonné : procedure ;
    Occup, nbAbon, nbNAbon : entier …… ;
    Abon, NAbon: condition
```

```
Procedure Entrer-Abonné
Début
 si Occup < N alors
   Occup := Occup + 1
 sinon
   ……
   wait(Abon)
 finsi
– 1
Fin
```

```
Procedure Entrer-Non-Abonné
 Début
   si Occup < N alors
      Occup := Occup + 1
   sinon
     ……..
     waitP(NAbon)
   finsi

 Fin
```

```
Procedure Sortir-Voiture
  Début
   si nbAbon > 0 alors
      nbAbon := nbAbon – 1
      signal (Abon)
   sinon
    si nbNAbon  > 0 alors
        nbNAbon  := nbNAbon

        signal (NAbon)
        ……
        ……
     Finsi
   Finsi
  Fin
```

Exercice 3

Soit N processus Pi (i=1..N) et un processus Ps. Les processus Pi (i=1..N) remplissent un tampon pouvant contenir M messages (M cases), un seul à la fois étant autorisé à déposer son message. Le processus Pi qui remplit la dernière case du tampon active le processus Ps qui fait alors l'impression de tous les messages déposés dans le tampon. Durant cette impression, les processus Pi (i=1..N) ne sont pas autorisés à accéder au tampon.

Ecrire les algorithmes des processus Pi (i=1..N) et Ps en utilisant :

a) Les sémaphores
b) Les régions critiques

7.4 Examen 4

Exercice 1 : Répondre en encerclant la réponse jugée juste. Une réponse juste = 0,25 point.

1. La notion d'exclusion mutuelle d'accès à une variable commune est basée sur la notion:
   a. D'exclusion mutuelle d'accès à une cellule mémoire b. De synchronisation des processus concurrents
   c. De processus dans un système d'exploitation d. La réponse juste n'est pas donnée
2. La synchronisation des processus concurrents pour un objet commun a pour but :
   a. D'établir un ordonnancement des exécutions des processus
   b. De maintenir la cohérence de l'objet commun
   c. De créer une priorité des processus pour l'accès à l'objet commun
   d. La réponse juste n'est pas donnée
3. La section critique peut être définie comme :
   a. Un processus en exécution b. Une séquence d'instructions en exécution
   c. Une séquence d'instructions en exécution utilisant une variable commune
   b. La réponse juste n'est pas donnée
4. Une Section Critique (SC) qui demande un temps d'exécution important est :
   a. Une bonne SC, car elle permet au processus de terminer son exécution
   b. Une mauvaise SC, car elle empêche les autres processus de s'exécuter
   c. Une mauvaise SC, car elle retarde la prise en compte des événements urgents
   d. La réponse juste n'est pas donnée
5. Le blocage d'un processus à l'intérieur d'une section critique risque :
   a. Le blocage de tous les processus concurrents à l'utilisation de l'objet commun
   b. Le blocage de tout le système c. a. ou b.
   d. La réponse juste n'est pas donnée
6. Les instructions de masquage/démasquage pour créer des sections critiques
   a. Peuvent être utilisées dans un système monoprocesseurs
   b. Peuvent être utilisées dans tout type de système d'exploitation
   c. Peuvent être utilisées seulement sur des processeurs qui possèdent ces instructions.
   d. La réponse juste n'est pas donnée

7. Pour synchroniser des processus dans un système multiprocesseurs, on peut utiliser :
   a. Les instructions de masquage/démasquage b. Les instructions de type Test and set
   c. Les instructions de type exchange d. a ou c
8. L'avantage des sémaphores pour créer des sections critiques est :
   a. Leur généralité d'utilisation à tout type de problème
   b. Leur structuration simple c. Leur simplicité d'utilisation
   d. La réponse juste n'est pas donnée
9. L'inconvénient de l'utilisation des sémaphores est :
   a. Leur gaspillage en temps b. Ils n'existent pas dans tous les systèmes
   c. Manque de structuration dans un programme
   d. La réponse juste n'est pas donnée
10. L'avantage des moniteurs pour synchroniser des processus concurrents est :
   a. Leur facilité d'utilisation b. L'évitement de l'interblocage des processus
   c. Ils sont présents dans tous les systèmes
   d. La réponse juste n'est pas donnée
11. L'inconvénient principal des moniteurs classiques (Hoare) lors de leur implémentation est :
   a. L'absence de réveil en chaine des processus en attente
   b. La primitive signal c. L'implémentation des variables de type condition
   d. La réponse juste n'est pas donnée
12. L'inconvénient majeur des régions critiques est :
   a. Le gaspillage de temps b. Le manque d'expressivité
   c. La difficulté d'utilisation
   d. La réponse juste n'est pas donnée
13. L'outil de synchronisation le mieux adapté pour une application temps réel monoprocesseur :
   a. Les moniteurs b. Les sémaphores
   c. Les régions critiques
   d. La réponse juste n'est pas donnée
14. L'avantage des expressions de chemin est :
   a. Leur simplicité d'utilisation
   b. Leur bonne structuration
   c. Leur absence de blocage
   d. La réponse juste n'est pas donnée
15. L'inconvénient des expressions de chemin réside dans :
   a. Leur manque de généralité d'utilisation
   b. Leur manque de disponibilité dans beaucoup de systèmes
   c. Leur coût en temps
   d. La réponse juste n'est pas donnée
16. Un deadlock se produit si
   a. Les 4 conditions sont fausses
   b. Au moins une condition est fausse
   c. Les 4 conditions sont vraies
   d. La réponse juste n'est pas donnée

Exercice 2

Soit un taxi collectif, de capacité C places, assurant le transport des clients du point x vers le point y.
Le taxi ne peut quitter x que s'il est chargé (nombre de passagers égal à C). Tout client qui arrive au point x se met en attente du taxi. Une fois que le taxi est au point x, les clients sont invités à monter dans le taxi. A l'arrivée au point y le chauffeur invite les passagers à descendre, une fois le taxi est vide il revient au point x et le cycle recommence toute la journée. On suppose qu'il y a un nombre multiple C de clients en attente au point x.
Voici quelques détails supplémentaires :

- Les clients devraient monter dans le taxi en appelant la procédure **Monter** et descendre du taxi (à l'arrivée à l'endroit y) en appelant la procédure **Descendre**.
- Le taxi devrait invoquer la procédure **Embarquer** lorsqu'il décide de charger et la procédure **Débarquer** lorsqu'il arrive à la destination y.
- Les clients ne peuvent monter dans le taxi que lorsque celui-ci ait invoqué **Embarquer.**
- Les clients ne peuvent en descendre jusqu'à ce que le taxi ait invoqué **Débarquer**.

Le travail du taxi et des clients se résume comme suit :

```
Taxi
Cycle
   Embarquer ()
  /* circuler de x vers y */
   Debarquer ()
 /* circuler de Y vers x */
Fcycle
```

```
Client
Arriver ()
Monter ()
 /* voyager de x vers y */
 Descendre ()
```

Un étudiant a proposé la solution suivante :

Var AT, PM, PT, m, PD: semaphore;
Nbc: entier init 0,

Procedure Arriver

Debut

P(AT)

Fin

Prcedure Monter

Debut

nbc := nbc+1

Si nbc=C alors

Fsi

Fin

Procedure Descendre

Debut

P(m)

nbc := nbc-1

V(PT)

Fin

Procedure Embarquer

Debut

V(AT)

Fin

Procedure Debarquer

Debut

V(PD)

Fin

Questions:

1) Compléter la solution proposée par l'étudiant sans oublier d'initialiser les sémaphores utilisés, afin de résoudre le problème posé.
2) Transformer la solution proposée en 1) en utilisant les moniteurs de Hoare
3) Si maintenant on a deux types de clients : certains sont des abonnés et d'autres non abonnés et que le taxi privilégie les clients abonnés pour monter dans le taxi avant les autres, proposer une solution en utilisant les régions critiques.

7.5 Examen 5

Exercice 1: Cet exercice peut être vu comme une extension du problème classique des producteurs consommateurs. En effet, chaque fois qu'un producteur a produit une information et l'a déposée dans le tampon, la case qu'occupe l'information ne sera libérée que lorsque tous les consommateurs l'auront consommée. On suppose que le nombre de consommateurs est fixé. Bien sûr, quand un consommateur aura réussi à consommer l'information d'une des cases du tampon circulaire, il pourra poursuivre son activité normale, c'est à dire, se préoccuper d'aller consommer la case suivante dès lors qu'elle contient une information résultant d'une production. De même pour un producteur, qui pourra continuer à produire si bien sûr il y a encore de la place libre. Donner une solution en utilisant les régions critiques.

Exercice 2 : Soit un groupe d'étudiants préparant un examen, les étudiants étudient tout en mangeant des pizza. Chaque étudiant exécute la boucle suivante : prendre un morceau de pizza; étudier en mangeant. Si un étudiant constate que la pizza est terminée (Pas de pizza disponible) il s'endort jusqu'à ce qu'une autre pizza arrive. Le premier étudiant qui désire manger et constate qu'il n'y a aucun morceau de pizza sur la table, il commande une autre avant de s'endormir. Chaque pizza est composée de S morceaux.

Question : Donner les algorithmes de synchronisation des processus étudiant et livreur de pizza en utilisant les sémaphores et seulement les variables suivantes.

```
Var mutex : sémaphore binaire ; prive1, prive2 : sémaphore privé ;
    Nb : entier init S ; F : booleen init vrai ;

Processus étudiant                    Processus livreur
Debut                                   debut
 Cycle                                  Cycle
  Prendre un morceau de pizza;            Préparer pizza
  Etudier ;                              Livrer pizza
 Fincycle                               Fincycle
Fin                                    Fin
```

Exercice 3 : Répondre en encerclant la lettre correspondant à **la réponse juste**. Une réponse juste = 0,5 point.

1. Les processus suivants partagent la variable X, initialisée à 5, en utilisant un sémaphore privé T :

```
Processus A          Processus B
int Y;               int Z;
A1: Y = X*2;         B1: P(T);
A2: X = Y;           B2: Z = X+1;
V(T);                X = Z;
```

Le nombre possible des valeurs différentes de X après l'exécution des processus A et B est :

a) 1
b) 2
c) 3
d) 4

2. L'exécution atomique des primitives P et V dans un système multi processeurs est réalisée par :

a) Le masquage des interruptions
b) L'instruction TAS
c) Un logiciel spécial
d) La réponse juste n'est pas donnée

3. Le problème observé dans la solution donnée en cours du modèle producteurs/consommateurs, à l'aide des moniteurs est :

a) L'exclusion mutuelle non assurée
b) L'interblocage des processus producteurs/consommateurs
c) L'absence de parallélisme entre producteurs/consommateurs
d) La réponse juste n'est pas donnée

4. Considérons un sémaphore *sem* initialisé à 10; 6 processus ont exécuté P(sem) et 4 processus ont exécuté V(sem). La valeur du sémaphore est alors :

a) 4 b) 6 c) 8 d) La réponse juste n'est pas donnée

5. Le moyen permettant à plusieurs processus d'utiliser correctement des ressources partagées de façon exclusive est :

a) La synchronisation
b) La section critique
c) Les variables partagées
d) La réponse juste n'est pas donnée

6. Les moniteurs se distinguent principalement des sémaphores par :
a) Leur facilité d'emploi
b) L'exécution automatique exclusive des procédures d'un moniteur
c) L'absence d'interblocage des processus du moniteur
d) La réponse juste n'est pas donnée

7. ……..……. est une technique qui peut être utilisée pour résoudre les conflits d'accès.
a) Section critique
b) Synchronisation
c) L'exclusion mutuelle
d) La réponse juste n'est pas donnée

8. La partie du code d'un processus nécessitant un accès exclusive aux ressources partagées est.
a) Section critique
b) Variables partagées
c) Appels des primitives P et V
d) La réponse juste n'est pas donnée

9. L'opération Test And Set s'exécute :
a) après un processus particulier
b) périodiquement
c) de façon atomique
d) La réponse juste n'est pas donnée

10. Lorsque la valeur d'un sémaphore est négative :
a) Elle représente le nombre de processus en attente sur ce sémaphore
b) elle n'est pas valide.
c) aucune opération ne peut être effectuée sur elle jusqu'à ce que V soit exécutée
d) La réponse juste n'est pas donnée

11. Le programme suivant est composé de 3 processus concurrents et 3 sémaphores initialisés comme suit :
S0 = 1, S1 = 0, S2 = 0.

```
Processus P0          Processus P1        Processus P2
Debutcycle            {                   {
{                     P(S1);              P(S1);
P(S0);                V(S0);              V(S0);
Printf( '0');         }                   }
V(S1);
V(S2);
Fincycle
}
```

Combien de fois P0 imprime 0 ?
A) au moins deux fois
b) exactement deux fois
c) exactement trois fois
d) exactement une fois

12. Chaque processus Pi, i = 0, 1, 2,……,9 est codé comme suit :

```
Cycle
  P(mutex)
 {Section Critique}
  V(mutex)
 Fincycle
```

Le code de P10 est identique aux autres sauf qu'il utilise V(mutex) à la place de P(mutex). Quel est le plus grand nombre de processus qui peuvent être à l'intérieur de la section critique à tout moment (mutex est initialisé à 1) ?
a) 1
b) 2

c) 3
d) La réponse juste n'est pas donnée

13. Soient deux processus P1 et P2 dont les codes sont comme suit :

```
Process P1                              Process P2
Cycle                                   Cycle
 {w1 = vrai;                           {w2 = vrai;
A : Si w2 == vrai alors aller à A        B : Si w1 == vrai alors aller à B
 {Section Critique};                     {Section Critique};
 w1 = faux;                              w2 = faux;
Fincycle}                               Fincycle}
```

Les variables w1 et w2 sont partagées et initialisés à faux. Laquelle des réponses suivantes est correcte ?
a) L'exclusion mutuelle est satisfaite
b) La solution présente un blocage des processus
c) Les processus entrent en section critique en alternance stricte
d) Il y a interblocage, mais l'exclusion mutuelle est assurée

14. La synchronisation des processus peut être réalisée :
a) Au niveau matériel
b) au niveau logiciel
c) au niveau matériel et logiciels
d) La réponse juste n'est pas donnée

15. Un moniteur est un module qui englobe
a) Des structures de données partagées
b) Une synchronisation des appels de procédures concurrentes
c) Des procédures qui manipulent une structure de données partagées
d) a), b) et c)

16. Les processus suivants partagent la variable X initialisée à 5:

```
Processus A          Processus B
int Y;               int Z;
A1: Y = X*2;         B1: Z = X+1;
A2: X = Y;           B2: X = Z;
```

Le nombre possible des valeurs différentes de X après exécution des processus A et B est :
a) 2 b) 3 c) 4 d) 8

7.6 Examen 6

Exercice 1 : Répondre en cochant la lettre correspondant à **la réponse juste**. Une réponse juste = 0,5 pts, fausse = -0,25

1. La synchronisation des processus est un mécanisme permettant à plusieurs processus:
a) Indépendants de partager des ressources de manière cohérente
b) Concurrents de partager des ressources de manière cohérente
c) Coopérants de manipuler des ressources sans interblocage
d) La réponse juste n'est pas donnée

2. Une section critique est une suite d'instruction qui :
a) Evite l'interblocage b) Manipule les ressources non partagées
c) Doit être toujours encadrée par les primitives P et V d'un sémaphore
d) La réponse juste n'est pas donnée

3. Soit un sémaphore *mutex* initialisé à 12; 8 processus ont exécuté P(*mutex*) et 3 autres ont exécuté V(*mutex*). Soit (p,r), où p est le nombre de processus utilisant les ressources et r le nombre de ressources disponibles. Le couple (p,r) correct est :
a) (6, 2) b) (4, 7) c) (5, 6) d) La réponse juste n'est pas donnée

4. Pour protéger une section critique, on utilise un sémaphore qui est :
a) Une solution hard dans le système
b) Un programme spécial dans le système
c) Une variable entière
d) La réponse juste n'est pas donnée

5. La condition nécessaire pour qu'il y ait un interblocage est:

a) L'exclusion mutuelle
b) Allocation et attente
c) Pas de réquisition de ressources
d) La réponse juste n'est pas donnée

6. Pour éviter l'interblocage, on exécute l'algorithme de Banquier:
a) A des intervalles de temps fixes
b) A chaque fois qu'une ressource est demandée
c) a) et b)
d) La réponse juste n'est pas donnée

7. Soit un système à P processus et R unités de ressources identiques. Si chaque processus peut demander au plus N unités de ressources, alors l'interblocage ne peut arriver si et seulement si :
a) $R < P*(N-1) + 1$ b) $R > P*(N-1) + 1$
c) $R \geq P*(N-1) + 1$ d) La réponse juste n'est pas donnée

8. L'inconvénient des régions critiques s'observe dans :
a) L'inefficacité dans le réveil des processus en attente b) Difficulté d'utilisation
c) Manque de structuration d) La réponse juste n'est pas donnée

9. Des processus concurrents peuvent se synchroniser en appelant des procédures d'un moniteur (Hoare). Un appel de la primitive de réveil *signal* permet:
a) De réveiller tous les processus bloqués par un wait
b) De ne réveiller qu'un processus bloqué par wait s'il y en a
c) De réveiller un processus bloqué s'il y en a et de bloquer éventuellement l'appelant
d) La réponse juste n'est pas donnée

10. Les procédures suivantes d'un moniteur sont appelées par deux processus (i et j) partageant la variable X initialisée à 2:

Var condition cond1: X=2; cond2 : X=4 ;

Procedure A()	Procedure B ()	Processus i	Processus j
int Y;	int Z;	……	……
wait (cond1);	wait (cond2);	A() ;	B() ;
Y = X*2;	Z = X+1;	……..	……..
X = Y;	X = Z;		

L'ordre d'exécution des processus est :
a) Processus i puis j b) Processus j puis i
c) Selon l'ordre d'arrivée d) La réponse juste n'est pas donnée

11. Les outils de synchronisation des processus concurrents tels que : sémaphores, les moniteurs, les régions critiques …. se basent essentiellement tous sur :
a) Les outils de synchronisation matériels
b) L'indivisibilité des accès aux cellules mémoire
c) Le verrouillage des bus mémoire d) La réponse juste n'est pas donnée

12. Le problème de section critique est résolu si :
a) Un seul processus manipule la ressource critique
b) Le processus ne peut être bloqué si la ressource est libre
c) l'attente est limitée d) La réponse juste n'est pas donnée

Exercice 2

Soit N processus, numéroté de 1 à N (N>1), qui arrivent tous en même temps pour exécuter:

```
while(true)
  {
    demander_tampon(i) ;
    //utiliser le tampon
    libérer_tampon(i) ;
  }
```

En utilisant **les sémaphores**, écrire les procédures **demander_tampon(i)** et **libérer_tampon(i)** de sorte qu'**un seul processus utilise le tampon à la fois** et en partant du principe que **les processus pairs sont plus prioritaires que les processus impairs**.

Exercice 3

Soient N processus P1, P2,...Pn concurrents pour l'utilisation exclusive d'un fichier F (ressource non partageable). Si F est libre, il peut être utilisé immédiatement autrement le processus demandeur doit attendre sa libération (par le processus détenteur). Si plusieurs processus sont en attente de F celui-ci sera attribué, dès sa libération, à un processus parmi ceux en attente selon la politique FIFO (first in first out). C'est-à-dire le premier arrivé bloqué (en attente) sera le premier réveillé.

Question : Donner la solution du gestionnaire du fichier F en utilisant les moniteurs de Hoare.

7.7 Examen 7

Exercice 1

1. Donner la différence entre un blocage et un interblocage des processus
2. Une section critique est une ……qui manipule……..…….
3. Les outils de synchronisation des processus concurrents tels que : ……,
   …… et …… se basent sur ……..
4. La synchronisation des processus est un mécanisme permettant à plusieurs processus …….. de partager ……de manière cohérente.
5. Dans les sémaphores, la primitive P est procédure ……alors que la primitive V est ……..……
6. La primitive signal, des moniteurs, permet soit de ……. le processus appelant et …….. un processus …….. ou un signal ……

Exercice 2

Soient trois processus concurrents P0, P1 et P2 qui communiquent au moyen de deux tampons T0 et T1 de même taille.

- P0 et P1 partagent le tampon T0
- P1 et P2 partagent le tampon T1

Le processus P0 se charge de lire du clavier des messages, à l'aide de la fonction **Mess-Lire ()**, qu'il traite avant de les déposer dans le tampon T0. Le traitement par P1 consiste à l'encrypter. Il est réalisé par la fonction **Mess-encrypter(msg).** Le processus P1 transfert directement les messages encryptés dans le tampon T1. Le processus P2 récupère les messages du tampon T1 pour les envoyer à un destinataire. L'envoi d'un message est réalisé par la fonction **Envoyer(msg)**.

Question : Complétez les pseudo codes suivants réalisant la communication entre P0 et P1 et entre P1 et P2:

```
Var mutex0, mutex1 : semaphore init 1 ;
    sem0, sem1: semaphore init 0;
```

```
Processus P0
debut
  Mess-lire()
  ……
 deposer(msg)
  ……...
  ……
fin
```

```
Processus P1
 debut
   ……..
   …….
   prelever (msg)
    …….
   Mess-encrypter(msg)
     ……
   deposer(msg)
     …….
      …….
 fin
```

```
Processus P2
  debut
     ……
     ……..
    prelever(msg)
      ……..
    Envoyer(msg)
  fin
```

Exercice 3

Soient N processus P1, P2,...Pn concurrents pour l'utilisation exclusive d'un fichier F (ressource non partageable). Si F est libre, il peut être utilisé immédiatement autrement le processus

demandeur doit attendre sa libération (par le processus détenteur). Si plusieurs processus sont en attente de F celui-ci sera attribué, dès sa libération, à un processus parmi ceux en attente selon la politique LIFO (last in first out). C'est-à-dire le dernier arrivé bloqué (en attente) sera le premier réveillé.

Question : Complétez les pseudo codes suivants réalisant le gestionnaire du fichier F en utilisant les moniteurs de Hoare.

Gestionnaire_Fichier_LIFO.Monitor ;

```
Const N =10 ;
Var occupe : Boolean ;
cond :array [1…N] of Condition ;
pile : array[1…N] of Integer;
sommet : Integer;
Procédure Demander (i :Integer)
Begin
   If occupe = false then
      …….. ;
  Else
     {
       Sommet ++;
       Pile[sommet] = i;
       ……..;
     }
End

Begin//Initialisation
……;
…….;
End;
```

```
Procédure Liberer( )
Var j:integer ;
Begin
     If sommet …….. then
       {
          j = pile[sommet];
          …… ;
          ……;
       }
     else
        …….. ;
End
```

7.8 Examen 8

Exercice 1

Soit l'allocateur de ressources gérant N ressources identiques et constitué des deux procédures : Allouer(i, n) (où i est l'identité du processus et n le nombre de ressources demandées par le processus i) et Liberer(n) (n est le nombre de ressources libérées par un processus). La procédure Liberer(n) doit réveiller seulement les processus qui seront effectivement servis (c'est-à-dire les processus bloqués dont la demande peut être satisfaite). Résoudre le problème en utilisant le Moniteur de Hoare.

Allocateur.Monitor

Exercice 2

Soient trois procédures P, Q, R manipulant des données communes et pouvant être appelées par des processus distincts. Pour des raisons sémantiques des services rendus, les procédures P, Q, R (s'exécutant chacune une fois) doivent respecter le schéma d'exécution suivant: (P --> Q --> R) autrement dit: L'exécution de P doit précéder celle de Q qui à son tour doit précéder celle de R.

Résoudre le problème en utilisant le Moniteur de Kessels.

7.9 Examen 9

Exercice 1

Soient N processus P1, P2, …, Pn concurrents pour l'utilisation de M imprimantes identiques du système (M<N). Les imprimantes sont demandées une à une. L'algorithme du gestionnaire sensé gérer l'affectation de ces imprimantes est le suivant :

Var S1, S2 : semaphore init 1;

Var sembloc : semaphore init 0;

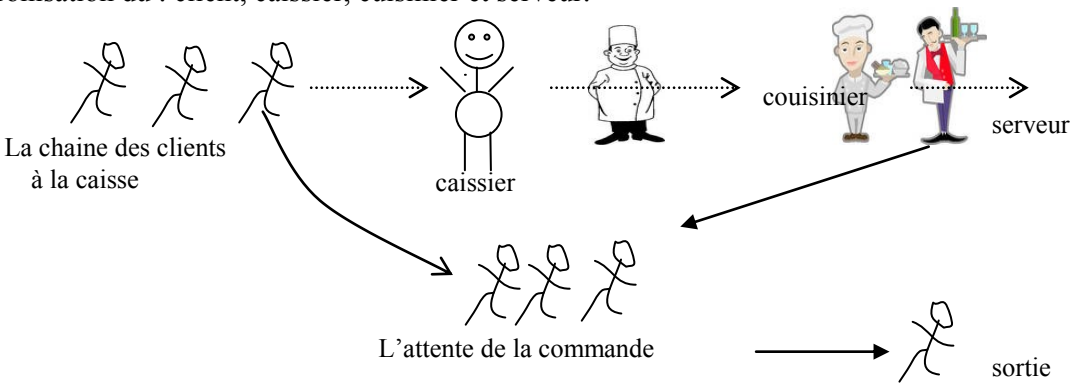
```
Var impdisp : integer init M;
Var nbprocbloc: interger init 0;

Procedure demander
Begin
  P(s1)
    If impdisp > 0 then impdisp := impdisp -1;
    else begin
           nbprocbloc : nbprocbloc + 1;
           P(sembloc); end;
   endif
  V(s1)
End
```

```
Procedure liberer
Begin
 P(s2)
    impdisp := impdisp + 1;
    If  nbprocbloc > 0 then begin
      nbprocbloc :=  nbprocbloc -1;
            v(sembloc); end;
    endif
  V(s2)
 End
```

Questions : Indiquez si ce gestionnaire est correct, sinon donnez la solution juste.

Exercice 2

Soit un restaurant Fast-food où les clients arrivent devant le caissier du restaurant et font leurs commandes une à une suivant l'ordre d'arrivée et attendent sa préparation. Le caissier traite les clients un à un (s'il y en a) sinon il se met en attente. Toute commande enregistrée (payée) est déposée par le caissier chez le cuisinier. Ce dernier prépare la commande et la remet au serveur qui doit la remettre au client en attente de sa commande. Ecrire le moniteur (classique) qui réalise la synchronisation du : client, caissier, cuisinier et serveur.



7.10 Examen 10

Exercice 1

Les procédures suivantes d'un moniteur sont appelées par deux processus P1 et P2 partageant la variable A.

```
Var cond1, cond2: condition; A: entier;
Procedure XX()
Debut
  Si A=2 alors
    wait (cond1);
   Finsi
  Signal(cond2);
   A = A+2;
 Fin
```

```
Procedure YY ()
 Debut
   Si A=0 alors
        wait (cond2);
  Finsi
   A = A*A;
   Fin
```

```
Processus P1
……
XX() ;
……..
```

```
Processus P2
……
YY() ;
……..
```

Initialisation
A= 0.

Question : En justifiant votre réponse, donnez la valeur finale de A.

Exercice 2

Soient deux processus partageant une variable A, afin de synchroniser ces 2 processus je propose la solution suivante.

```
Var Avous : entier ;
    Avous  = 1;
```

Processus1
debut
test: ***Tantque*** Avous = 2 ***faire***
allera test
Fintantque
A++
Avous := 2;
fin

Processus2
debut
test: ***Tantque*** Avous = 1 ***faire***
allera test
Fintantque
A =A*2;
Avous := 1;
fin

Question : Déterminez le problème de cette solution en justifiant votre réponse.

Exercice 3

Soient trois procédures d'un moniteur appelées par deux processus P1 et P2 partageant la variable A.

Var condition cond1 : ……., cond2 : ……..; cond3 : ……. ; A: entier;

Procedure B()	Procedure C()	Procedure D()
Debut	Debut	Debut
wait (cond1);	wait (cond2);	wait (cond2);
A = A+2;	A = A*5;	A = A*A+3
Fin	Fin	Fin

Initialisation
A= 1.

Question : Complétez les trois procédures précédentes sans ajouter de variables afin que la valeur finale de A soit 30.

13. Soient un producteur produisant des messages et un consommateur prélevant ces messages qui sont déposés dans un tampon. Complétez la solution suivante qui permet la bonne gestion du tampon tout en assurant la cohérence des messages déposés dans le tampon et en évitant la perte des messages et le prélèvement redondant des messages.

Var …….

Procedure producteur
begin
cycle
< Produire un message >;
deposer(msg);
endcycle
end

Procedure Consommateur
begin
cycle
prelever(msg);
< consommer le message >;
endcycle
end

Exercice 4

Simuler un sémaphore à l'aide du moniteur de Kessels

Sem.***Monitor***
begin
var comptsem : ***integer*** ;
condition cond: ……..
procedure P
begin
……..

end
procedure V
begin
……..

end
Initialisation
……..

end monitor.

7.11 Examen 11

Exercice 1
Soient cinq (5) processus P1, P2, P3, P4 et P5 qui font les calculs suivants : S1= A*2 (par P2), S2=S1*2 (par P1), S3= S1/2 (par P3), S4= S3+4 (par P4), S5= S2+S3 (par P5) ; donner une solution de synchronisation de ces 5 processus en utilisant seulement trois sémaphores.

Exercice 2
On considère N processus Pi et un processus Maître comme suit :

Processus Pi	Processus Maître
Début PA ; PB ; Fin	Début MA ; MB ; Fin

- Les N processus Pi et le processus Maître s'exécutent en parallèle.
- Chaque processus Pi exécute la partie d'instructions PA et se bloque.
- Après avoir terminé la partie d'instructions MA, le processus Maître attend que tous les processus Pi aient terminé chacun sa partie PA ; il poursuit alors l'exécution de la partie MB.
- Une fois la partie MB terminée, le processus Maître libère tous les processus Pi bloqués, qui peuvent alors continuer leur exécution.

Question : Proposez une solution de synchronisation des processus Pi et processus Maître en utilisant les sémaphores.

Exercice 3
Soient trois processus utilisant trois ressources, une solution de synchronisation de ces processus donnée comme suit :

Var sem1, sem2, sem3 : sémaphore binaire;		
Processus P1 Debut P(sem2) ; utiliser ressource R2 ; P(sem3) ; utiliser ressource R3 ; V(sem3) ; V(sem2) ; Fin	Processus P2 debut P(sem3) ; utiliser ressource R3 ; P(sem1) ; utiliser ressource R1 ; V(sem1) ; V(sem3) ; Fin	Processus P3 debut P(sem1) ; utiliser ressource R1 ; P(sem2) ; utiliser ressource R2 ; V(sem2) ; V(sem1) ; Fin

Questions : Une réponse non justifiée n'est pas considérée

1) Donner un chemin d'exécution des trois processus qui mène à leurs terminaisons (fin d'exécution des 3), justifier votre réponse.
2) Donner un chemin d'exécution des trois processus qui ne permet pas la terminaison des trois processus. Quel est le problème ? Justifier votre réponse.
3) Donner une solution au problème précédent en corrigeant la solution proposée ci-dessus.

Exercice 4

Considérons une variante du problème producteur/consommateur (vu en cours m producteurs, p consommateurs et n cases). Nous avons un seul producteur, n consommateurs et un tampon à n cases. Le producteur dépose le même message produit dans toutes cases. Le consommateur i consomme (prélève) le message déposé dans la case i. Le producteur et les consommateurs font le même travail avec plusieurs messages. Donner une solution de synchronisation des processus, en utilisant les moniteurs, dont la solution à l'aide des sémaphores est la suivante :

```
Var Vide : tab[1, N] de sémaphore binaire ;
    Mutex : tab[1..N] de sémaphore privé;
    Tampon : tab[1..N] de caractères ;
```

Producteur	Consommateur(i)
int ip =0 ; Cycle <Produire un message m>; pour j=0 à n- 1 faire P(Vide[j]) ; Tampon[j]=m; V(Mutex[j]) ; Finpour Fincycle	int ic =0 ; Cycle P(Mutex[i]) ; m = Tampon[i]; V(Vide[i]) ; Fincycle

7.12 Examen 12

Exercice 1

Remplacer les pointillés par les termes qui conviennent.

…….. est protégée au moyen de ……, ou de …… permettant de résoudre le problème des accès concurrents à une ressource partagée par plusieurs processus.

La synchronisation des processus concurrents pour une ressource partagée a pour but de maintenir sa ……

Le blocage d'un processus à l'intérieur …….. entraine le blocage des …….. à l'utilisation de l'objet commun.

Parmi les solutions matérielles de l'exclusion mutuelle, les Verrous assurent …….. par rapport à ……

Lorsque la valeur d'un sémaphore est positive, elle représente le nombre de ……

L'inconvénient principal des moniteurs de Hoare lors de leur implémentation est ……

P et V sont des …… plutôt que des procédures car elles sont ……

L'interbocage se produit lorsqu'il y a une …… entre les processus concurrents.

Le graphe d'allocation des ressources n'est pas aussi fiable que …….. lorsque les ressources partagées sont en plusieurs exemplaires.

Un système est dans un …… s'il peut allouer des ressources à chaque processus dans un certain ordre tout en évitant un interbocage.

Parmi les outils de synchronisation vus en cours ……..… constituent le mécanisme de synchronisation le plus compliqué à utiliser.

Exercice 2

On considère un fichier F qui peut être utilisé par deux classes de processus A et B. Le nombre d'utilisations simultanées est illimité. Par contre les deux classes A et B sont mutuellement exclusives. Et de plus, on considère que les processus de la classe B sont plus prioritaires que ceux de A pour l'accès au fichier F.

Questions

1) Compléter la solution A) permettant de résoudre ce problème à l'aide des sémaphores.
2) Compléter la solution B) permettant de résoudre ce problème à l'aide des Moniteurs de Hoare.
3) Quel est le problème de la solution B) ? Justifier votre réponse.

A) Solution à l'aide des sémaphores :

Var nA , nB, nBatt : entier init 0 ;

nA, nB et nBatt représentent respectivement le nombre de processus classe A, de classe B et le nombre de processus classe B en attente.

S1, S2, mutex1, mutex2 : sémaphore binaire ;

Processus classe A	Processus classe B
Debut P(S2); P(mutex1) ; …….; Si nA=1 alors …….; Fsi …….; V(mutex1); <Utiliser le fichier F> P(mutex1) ; …….; Si nA =0 alors	Debut P(mutex2) ; …….; si nBatt=1 alors …….; Fsi nB ++; Si nB = 1 alors …….; Fsi …….; <Utiliser le fichier F>

V(S1) ; Fsi V(mutex1); Fin	P(mutex2) ; …….; nBatt -- ; Si nB =0 alors …….; Fsi V(mutex2) ; Fin

B) Solution à l'aide des moniteurs de Hoare :

LecA-B.Monitor

Var nA, nB, nBatt: ***entier***; lect_A, lect_B: *condition*;

Proc-A, Proc-B : Procedure ;

Procedure Proc-A
debut
Si …….. alors
wait(lect_A);
……..;
Fsi
nA ++;
< Utiliser le fichier F>
nA --;
Si …….. alors
……..;
Fsi
Fin

Procedure Proc-B
Debut
……. ;
Si …….. alors
…….. ; …….. ;
Fsi
nB++ ;
< Utiliser le fichier F>
……..…….. ;
…….. ;
Si …….. alors
……. ;
Fsi
Fin

Initialisation

……

FinMoniteur

8. Conclusion

Une machine ou ordinateur est structurellement composée de deux couches :

- Une couche physique rassemblant les différents circuits électroniques tels que : Le CPU, la mémoire centrale (RAM), les chips d'entrées/sorties, carte graphique, carte réseaux, etc.

- Une couche logique formant le logiciel qui est responsable du fonctionnement spécifique et adéquat de la machine globale. C'est cette couche que l'on désigne communément par système d'exploitation. Comme la couche physique, la couche

logicielle est aussi composée de module logiciels tels que : La gestion des processus, la gestion de la mémoire centrale, la gestion des entrées/sorties, la gestion des fichiers, etc.

Un système d'exploitation de type généraliste est conçu pour prendre en charge l'exécution de diverses applications des usagers auxquels il doit assurer une sureté de fonctionnement appropriée. C'est par le biais de celle-ci que les usagers accordent une confiance justifiée aux résultats issus des exécutions de leurs applications sous le contrôle du système hôte.

En fait, toute exécution d'une application correspond à l'exécution d'un ensemble de processus (ou threads) engendrés par cette application. Comme le nombre de ressources, à point d'accès unique, dans un système est beaucoup moins important que celui des processus qui les utilisent, il se crée alors des relations de concurrence entre ces processus. Lorsque la gestion de cette concurrence n'est pas minutieusement contrôlée, il se produit alors une incohérence de l'utilisation des ressources qui conduirait à un résultat erroné. Dans les systèmes critiques, un tel résultat erroné pourrait être à l'origine d'une catastrophe. D'où l'importance de ce cours destiné aux étudiants qui suivent le module SE2 ou tout autre développeur d'applications informatiques à processus concurrents nécessitant un niveau de sureté important.

BIBLIOGRAPHIE

1. André F. et al. "Synchronisation de programmes parallèles", Dunod,1983.
2. BenAri M. " Processus concurrents, introduction à la programmation parallèle", ed. masson, 1986.
3. BenAri M. `` Principles of Concurrent and Distributed Programming , Prentice Hall 1989
4. Brinch hansen " Operating system principles " ed. Prentice Hall, 1973.
5. Crocus " Systèmes d'exploitation des ordinateurs"2° ed. dunod 77.
6. Dijkstra E.W." The structure of the THE multiprogramming system" cacm, 11,5, 1968.
7. Dijkstra E.W. " A discipline of programming " Prentice Hall, 1986.
8. Habermann A.N., R.H. Campbell "The specification of process synchronization by path expressions " coll. sur les aspects théoriques et pratiques des systèms d'exploitation, IRIA, Paris 1974.
9. Hoare C.A.R. " Monitors: an operating system structuring concept, cacm 17,10, 1974.
10. Howard J.H. " Proving monitors, cacm 19,5, 1976.
11. Kessels J.L.W. " An alternative to event queues for synchronization in monitors cacm, 19,5, 1977.
12. Krakowiak S. " principes des systèmes d'exploitation"2° ed. dunod 1987.
13. Lhermitte CI. " Les systèmes d'exploitation: structure et concepts fondamentaux, ed. masson 1985.
14. Madnick S.E, J.J Donovan" Operating systems principles ", ed. Masson, 74.
15. Mossière J. " Méthodes pour l'écriture des systèmes d'exploitation " thèse d'état INP Grenoble 1977.
16. Raynal M. " Une analyse de la spécification de la coopération entre processus par variables partagées". TSI, 1,3, 1982.
17. Robert P., Verjus P. " Towards autonomous description of synchronization modules"; proc. IFIP Congress aug. 1977.
18. Shaw A.C. " The logical design of operating systems " ed. Prentice hall 1974.
19. Silberschatz A. Et al. `` Oparating System concepts , 3rd ed. Addison-Wesley, 1992
20. Tanenbaum A.S. " Operating systems: Design and Implementation" prentice hall 1987.
21. Tsichritzis D.C., Bernstein P.A." Operating systems " ed. Computer science and applied Mathematics, 1974.
22. William Stallings, Operating Systems: Internals and Design Principles, Sixth Edition Prentice Hall 2008
23. Stallings, William.Operating systems: internals and design principles 9th edition 2018

24. Andrew S. Tanenbaum, Albert S. Woodhull, Operating Systems: Design and Implementation,3rd edition 2006
25. Andrew S. Tanenbaum, Herbert BOS,Modern Operating Systems, Fifth Edition 2023
26. Chakraborty Pranabananda, Operating systems: Evolutionary concepts and modern design principles, First edition, [2024]