CHAPITRE 1 :

LA PROGRAMMATION CONCURRENTE

CHAPITRE 1

1. LA PROGRAMMATION CONCURRENTE

Références:
Chapitres 5 et 6 de William Stallings, **OPERATING SYSTEMS:** ***INTERNALS AND DESIGN PRINCIPLES***, 3e ed., Prentice Hall, 1998, isbn: 0-13-887407-7.

Chapitres 6 et 7 de Silberschatz et Galvin, **OPERATING SYSTEM CONCEPTS**, 4e ed., Addison Wesley, 1994, isbn: 0-201-50480-4; 5e ed., 1998, isbn: 0-201-59113-8.

1.1. EXÉCUTION CONCURRENTE ET SYNCHRONISATION

1.1.1. Séquence d'instruction et interrelation:

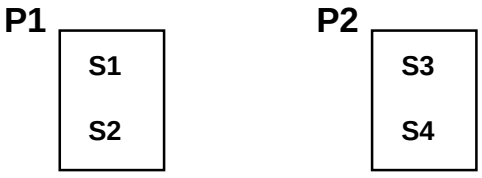
Un programme est une séquence d'instructions et le résultat que doit fournir un programme est directement relié à l'ordre particulier dans lequel les instructions sont exécutées. Une manière de considérer l'exécution d'un programme est de se placer au niveau du processeur. Il s'agit de regarder la séquence d'instructions machines qui est soumise au processeur. La séquence implique un aspect temporel. Chaque instruction est exécutée à un moment précis. Le résultat est déterminé par la suite d'instructions. La suite d'instructions constitue un profil ou une signature du programme. Un programme correct doit être reproductible et donc posséder une signature ou profil particulier et distinctif.

Lorsque deux programmes s'exécutent en même temps, en parallèle, nous pouvons considérer que nous avons deux séquences d'instructions parallèles. Cependant, il se peut que le rythme de progression des séquences ne soit pas constant dans le temps. A un moment donné, une séquence peut aller plus rapidement et plus tard ralentir. Dans un système complexe, plusieurs causes peuvent expliquer un tel changement de rythme. La mémoire virtuelle peut impliquer un délai variable selon que le contenu demandé est déjà en mémoire ou non. Le temps d'accès à une information sur disque est variable.

PROCESSUS CONCURRENT

INTERRELATION

→ **Considérer toutes les séquences (états) possibles - aspect temporel**



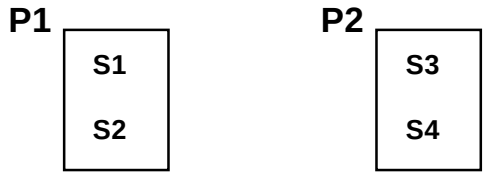
séquences possibles

→ S_1 → S_2 → S_3 → S_4
→ S_1 → S_3 → S_2 → S_4
→ S_1 → S_3 → S_4 → S_2
→ S_3 → S_1 → S_2 → S_4
→ S_3 → S_1 → S_4 → S_2
→ S_3 → S_4 → S_1 → S_2

Synchronisation → réduction des séquences possibles
Est-ce la bonne réduction ?
Pouvez-vous imaginer toutes les séquences ?

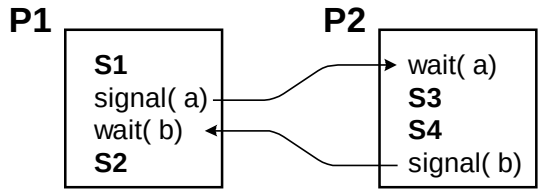
PROCESSUS CONCURRENT

Interrelation → considérer toutes les séquences possibles



Synchronisation → réduction des séquences

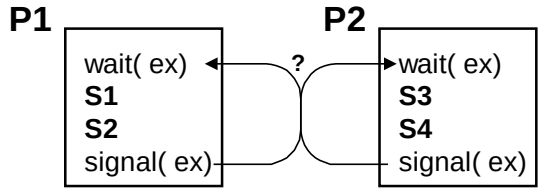
Sémaphore a = 0, b = 0 ;



une seule séquence possible :
→ S_1 → S_3 → S_4 → S_2

SECTION CRITIQUE:

Sémaphore ex = 1 ;



La variation dans la progression des programmes est sans importance si les deux programmes sont indépendants. C'est-à-dire que les résultats et la progression d'un programme ne sont pas affectés par les instructions exécutées par l'autre programme. Cependant, si les deux programmes sont dépendants, il faut préciser si une instruction d'un programme a eu lieu avant ou après une instruction de l'autre programme. Il faut alors tenir compte de la progression stricte dans le temps des deux programmes. On peut ainsi être amené à combiner les deux séquences en une seule, tenant compte de la progression relative des deux séquences. La séquence produite va consister en un entrelacement des deux séquences originales. Il nous faut considérer tous les cas d'entrelacements possibles.

Pour des programmes indépendants, tous ces entrelacements sont acceptables. Cependant, pour des programmes dépendants, seulement quelques uns de ces cas sont acceptables. Les autres doivent être évités, car ils conduisent à des résultats erronés ou à la catastrophe. Dans la programmation de programmes concurrents (et, sous-entendu, dépendants), il faut introduire des mécanismes qui permettent de synchroniser les processus et d'éliminer les séquences catastrophiques. La synchronisation implique que la progression d'un processus est éventuellement contrôlée par un autre. Cela implique l'arrêt éventuel d'un processus.

La coopération entre processus peut prendre la forme de

1) échange d'information (par exemple, par mémoire partagée ou par message)
2) synchronisation (par sémaphore ou moniteur)
3) et comme un cas particulier mais fréquent, la section critique. Ici, on dispose d'une mémoire commune (ou autres ressources) mais on doit se synchroniser pour assurer l'intégrité des données.

Même sur une machine avec un seul processeur, l'exécution concurrente de processus nous force à considérer tous les entrelacements possibles. En effet, on ne sait pas à quel moment particulier un processus va être interrompu pour permettre la reprise d'un autre processus. Le hasard peut faire en sorte qu'on obtienne n'importe laquelle des séquences possibles. Le besoin de synchronisation est toujours présent. Cependant, la présence d'un seul processus facilite l'implantation des mécanismes de synchronisation. Le noyau s'en occupe et s'assure de ne pas être interrompu ou perturbé dans son travail.

Il faut aussi considérer la granularité des instructions. Une instruction dans un programme Pascal ou C est en effet constituée de plusieurs instructions machines. C'est l'ordre dans lequel la machine reçoit ses instructions qu'il faut considérer. La granularité de la synchronisation se situe donc au niveau des instructions machines.

1.1.2. Section critique

La section critique est un cas particulier de synchronisation. Elle implique:

1) Des données (zone mémoire) communes utilisées par plusieurs programmes.
2) La section critique: une parti de code dans un programme qui accède ou utilise les données communes.

Pour assurer l'intégrité des données, un seul programme doit exécuter dans sa section critique. D'où un problème d'exclusion mutuelle. (Exclusion mutuelle = un seul à la fois) et (section critique = exclusion mutuelle).

Un exemple

Pour illustrer le problème de la section critique, nous pouvons examiner deux processus P0 et P1 qui produiront des résultats corrects quand ils sont exécutés séparément, mais qui peuvent à l'occasion produire des résultats erronés quand exécutés en concurrence. La Figure 1 présente le cas de deux processus qui utilisent une variable commune, x. Le processus P0 veut incrémenter x en exécutant l'instruction "x := x + 1", tandis que P1 décrémente x en exécutant "x := x - 1". Les sections critiques sont donc les instructions "x := x + 1" et "x := x - 1". On obtient un résultat correct seulement si un seul des processus exécute l'instruction critique à la fois. Si la valeur de x est initialement 5, alors l'exécution simultanée des deux instructions peut produire une valeur de 4, 5 ou 6 pour x. Naturellement, le résultat correct est 5. Ce résultat est obtenu, à coup sur, si les deux instructions sont exécutées une après l'autre.

Regardons comment une valeur incorrecte de x peut être obtenue. Il faut noter que l'instruction "x := x + 1" est traduit en langage machine par trois instructions:

```
reg_0 := x ;
reg_0 := reg_0 + 1 ;
x := reg_0 ;
```

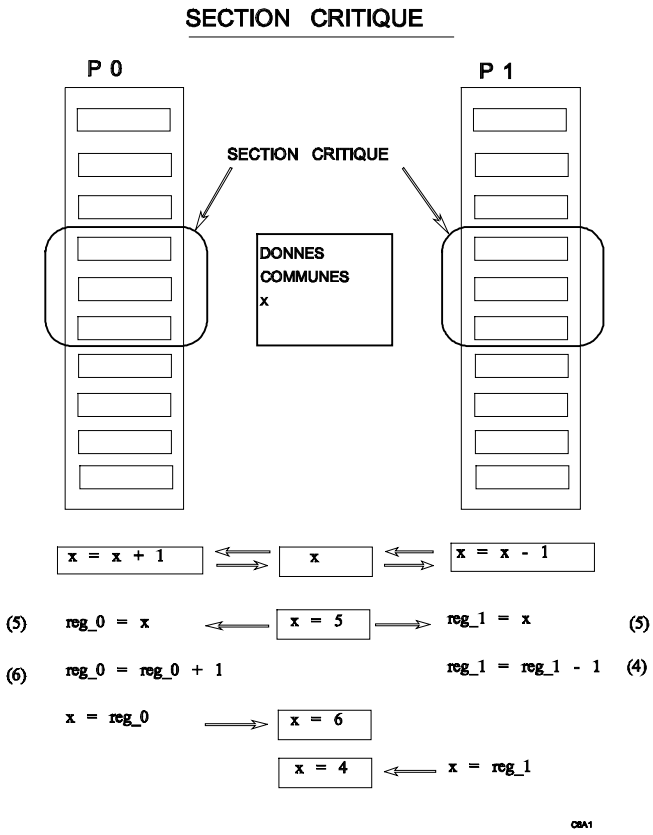


Figure 1 : Un exemple de section critique

ou reg_0 est un registre interne du processeur. De même, l'instruction "x := x - 1" est traduite par

```
reg_1 := x ;
reg_1 := reg_1 - 1 ;
x := reg_1 ;
```

ou reg_1 un autre registre interne. (En fait, reg_0 et reg_1 peuvent être le même registre. Il faut alors se rappeler que le contenu des registres est sauvegardé et restaurer par les routines d'allocation du CPU: le gestionnaire d'interruption et le commutateur de contexte.)

L'exécution des instructions de haut niveau en concurrence va se traduire par une séquence de six instructions assembler (de bas niveau). Pour un même processus, les instructions assembler doivent être exécutées dans l'ordre donné. Par contre, l'exécution concurrente indique qu'à tout moment, le CPU peut passer d'un processus à l'autre. On

pourrait donc ainsi obtenir les deux séquences suivantes:

a) t_0: P0 exécute reg_0 := x → reg_0 = 5
t_1: P0 exécute reg_0 := reg_0 + 1 → reg_0 = 6
t_2: P1 exécute reg_1 := x → reg_1 = 5
t_3: P1 exécute reg_1 := reg_1 - 1 → reg_1 = 4
t_4: P1 exécute x := reg_1 → x = 4
t_5: P0 exécute x := reg_0 → x = 6

le résultat final est x = 6.

b) t_0: P0 exécute reg_0 := x → reg_0 = 5
t_1: P0 exécute reg_0 := reg_0 + 1 → reg_0 = 6
t_2: P1 exécute reg_1 := x → reg_1 = 5
t_3: P0 exécute x := reg_0 → x = 6
t_4: P1 exécute reg_1 := reg_1 - 1 → reg_1 = 4
t_5: P1 exécute x := reg_1 → x = 4

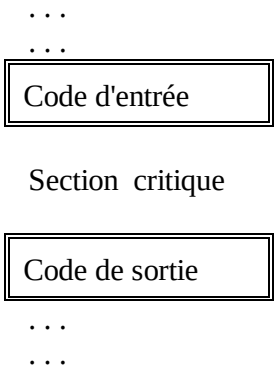
le résultat final est x = 4.

Dans ces deux cas, nous n'obtenons pas la bonne réponse qui est x = 5. Nous obtenons une réponse incorrecte parce que les deux processus manipulent la variable x en même temps. Nous devons donc nous assurer qu'un seul processus à la fois manipule la variable commune. Ceci constitue le problème de la section critique.

Problème de la section critique

Considérons un système composé de *n* processus qui opèrent en coopération $\{P_1, P_2, \ldots P_n\}$. Chaque processus possède une partie de code, appelée section critique, où le processus doit partager une ressource commune. La section critique peut consister par exemple par la lecture d'une variable commune, la mise à jour d'une table, l'écriture dans un fichier. Le point important est que lorsque d'un processus exécute sa section critique, aucun autre processus n'est autorisé à exécuter le code de sa section critique. Donc, l'exécution de la section critique est *mutuellement exclusive*. C'est à dire qu'un seul processus peut exécuter la section critique à la fois.

Il faut donc construire un protocole auquel va se conformer tous les processus pour assurer l'exclusion mutuelle. Ce protocole implique que chaque processus doit demander l'autorisation (la permission) pour rentrer dans sa section critique. Il faut donc ajouter du code dans le programme, juste avant la section critique, pour implanter cette requête. Ce code est désigné comme l'*entrée* de la section critique. De même, la section critique peut être suivie par du code ajouté correspondant à la *sortie* de la section.



Ce protocole ou code d'entrée et de sortie ne laisse passer qu'un processus à la fois, il bloque les autres. Ceci doit donc être supporté par les mécanismes appropriés pour la prise de décision et pour suspendre les processus.

Il faut cependant s'assurer que la suspension de processus ne paralysera pas l'ensemble. Pour cela, toute solution au problème d'exclusion mutuelle doit satisfaire trois conditions:

Exclusion mutuelle: Si un processus s'exécute dans sa section critique, alors aucun autre processus ne doit s'exécuter dans sa section critique.

Progression (ou pas d'interblocage): Si aucun processus n'exécute dans sa section critique et s'il y a quelques processus qui veulent entrer dans la section critique, alors seulement les processus exécutant le code d'entrée ou de sortie *(i.e., n'exécutant pas une partie du code non reliée à la section critique)* peuvent prendre par à la décision, et cette décision ne peut être retardée indéfiniment. La décision vise à déterminer quel processus va entrer dans sa section critique.

Attente bornée (ou pas de famine): Lorsqu'un processus demande d'entrer dans sa section critique, il doit y avoir une borne (valeur limite) sur le nombre de fois que les autres processus sont autorisés à entrer dans leurs

sections critiques avant qu'il n'obtienne finalement l'autorisation.

Il est entendu que chaque processus s'exécute à une vitesse non-nulle. Cependant, on ne peut faire aucune supposition concernant la vitesse *relative* des *n* processus.

Opération atomique

Le problème de la section critique peut être relié à l'*atomicité* des opérations ou instructions. Une opération est dite *atomique* si son exécution peut être vu comme prenant un temps nul. C'est à dire, si elle est exécutée complètement ou pas du tout, et si son résultat ne peut être affecté par ce qui s'est produit après son début. Ainsi, dans l'exemple ci-haut, si les instructions "x = x + 1" et "x = x - 1" étaient des opérations atomiques, la valeur de x serait toujours correcte et il n'y aurait pas de problème de section critique. Les instructions de base (assembler) d'un processeur peuvent généralement être considérés comme des opérations atomiques. Lorsqu'amorcé, une instruction s'exécute complètement et sans modification jusqu'à sa fin. En particulier, les instructions de base comme *load*, *store* et *test* du processeur sont considérées atomiques. Ainsi, si deux de ces instructions sont exécutées "simultanément", le résultat est équivalent à leur exécution séquentielle selon un ordre quelconque. Donc, si un *load* et un *store* sont exécutés simultanément, l'instruction *load* prendra soit l'ancienne valeur soit la nouvelle valeur, mais jamais une combinaison quelconque des deux.

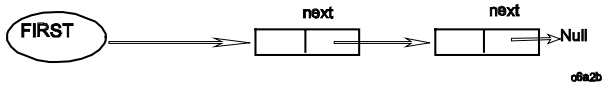
Dans le cas de l'exécution concurrente sur un seul processeur, l'exécution atomique d'une opération signifie que le transfert du CPU peut se produire seulement avant ou après l'opération, mais non durant son exécution. Par exemple, lorsqu'un processeur est en train d'exécuter une instruction assembler, il ne peut alors être interrompu par un signal d'interruption. Le processeur traitera l'interruption seulement quand il aura fini d'exécuter l'instruction. Certains processeurs ont des instructions complexes qui demandent beaucoup de temps d'exécution (plusieurs cycles d'horloge), par exemple, une instruction de copie de (longue) chaîne de caractères. Pour assurer une réponse rapide aux interruptions, tout en maintenant l'atomicité de l'opération, on a implanté un mécanisme de retour en arrière pour revenir au même état qu'avant l'exécution de l'instruction. L'interruption peut alors être traitée, et lorsqu'on retourne au processus suspendu, on recommence la longue instruction atomique.

Course critique

Considérons deux processus qui exécutent les mêmes instructions. On dit qu'il y a une course critique si le résultat dépend du fait qu'un processus exécute une instruction particulière avant l'autre. D'une manière plus générale, il y a une course critique si le résultat dépend du rythme de progression des différents processus impliqués. En générale, il faut éviter ces situations. La partie de code impliquée dans la course critique constitue donc, en fait, une section critique. On évite la course critique en assurant qu'un seul processus à la fois n'exécute le code.

Un exemple: liste chaînée

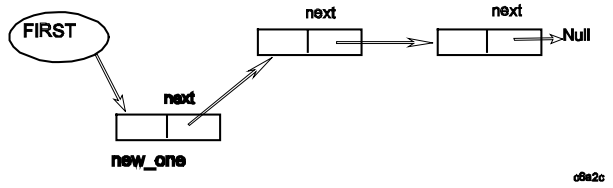
Dans les systèmes d'exploitation, on utilise souvent des files d'attente ou queue, par exemple, la queue des processus prêts à être exécutés ou les queues de requête d'entrée/sortie. Pour assurer l'intégrité des données, certaines opérations doivent être exécutées en exclusion mutuelle. Considérons, par exemple, le cas d'un ajout d'un élément dans une liste chaînée simple. Soit *node* une structure décrivant chacun des éléments de la liste. Cette structure possède un champ *next* (node.next) qui est un pointeur sur l'élément suivant de la liste. La variable *first* est un pointeur sur le premier élément de la liste.



L'ajout d'un élément *new_one* dans la liste implique donc les deux instructions suivantes:

a) new_one.next := first
b) first := address(new_one)

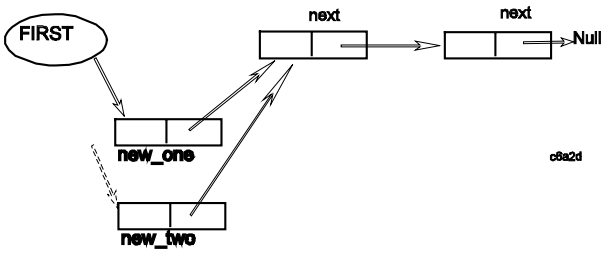
Après ces deux instructions, nous avons la nouvelle liste:



Supposons qu'en même temps que *new_one* est ajouté, un autre processus essaie d'ajouter l'élément *new_two* en exécutant les instructions:

c) new_two.next := first
d) first := address(new_two)

Supposons que les instructions a) et c) sont exécutées simultanément. Supposons maintenant que l'instruction d) est exécutée, placant l'adresse de *new_two* dans *first*; suivit immédiatement par l'instruction b) qui rajuste la valeur de *first* à l'adresse de *new_one*. La structure de la liste n'est alors plus consistante, avec un arrangement incorrect des pointeurs. Malheureusement, l'élément *new_two* est ici perdu; il ne peut pas être trouvé en suivant les liens à partir de *first*. Ceci serait un désastre si, par exemple, l'élément *new_two* représente un processus qui doit être exécuté.



1.1.3. La programmation concurrente

La programmation concurrente est un problème difficile. Elle fait partie des domaines frontières de l'informatique. L'avancement de l'informatique passe par l'exploration et la maîtrise de ces domaines.

Le succès de l'informatique est basé sur un équilibre judicieux entre la liberté et les contraintes. Un langage impose des contraintes. Il faut suivre certaines règles, syntaxes ou conventions si on désire être compris. A l'intérieur de ses règles, elle donne cependant une liberté d'expression. Il faut une liberté suffisante pour pouvoir dire des choses intéressantes. Pas assez de règles et trop de liberté implique malheureusement qu'il faut en dire trop pour être sûr d'être bien compris. Il est donc important de placer les règles et les libertés aux bons endroits, pour faire en sorte que l'on puisse dire ce qui est important à un domaine sans que cela soit trop compliqué.

L'informatique est née avec la programmation séquentielle. L'utilisation d'instruction de répétition (boucle) et de branchement (if) enrichit le pouvoir d'expression. Quelques décennies de recherche ont permis de découvrir et d'exploiter toutes les possibilités de la programmation séquentielle. Cependant, il y a des domaines où la programmation séquentielle est insuffisante ou non appropriée. La programmation séquentielle repose sur une isolation de chacun des programmes. Il faut maintenant considérer la coopération entre programmes.

La programmation concurrente est difficile. Il convient d'essayer d'esquisser un cadre de référence ou de présenter certains aspects qui doivent être considérés:

- vérifier que le programme est correct
- mécanisme de coopération (synchronisation)
- performance et/ou contraintes d'implantation

Un aspect particulièrement ardu de la programmation concurrente est de s'assurer que le ou les programmes sont corrects, c'est-à-dire qu'ils fournissent toujours les résultats escomptés. Il y a deux phases dans la vérification:

1) définir ce qu'est le comportement correct.
2) vérifier que les programmes (déjà écrits ou qu'on est en train d'écrire) suivent ce comportement correct.

La phase 1) s'apparente beaucoup de la spécification. Pour la programmation concurrente, c'est une phase qui n'est pas facile. On devrait être capable de dire pour chaque instruction des programmes qu'elles sont les conditions qui doivent

être satisfaites pour que l'exécution soit correcte (ou conforme au comportement correct). L'utilisation d'assertion entre chaque ligne des programmes peut être une manière de préciser le comportement correct.

Pour l'implantation (la programmation), il faut utiliser des mécanismes de synchronisation (de coopération) tels que les sémaphores ou les moniteurs. Il faut s'assurer que les mécanismes permettent de respecter le comportement correct, c'est-à-dire de respecter les assertions.

On peut donc concevoir le développement de programmes concurrents comme un processus de transformation du code des programmes:

1) écrire l'ossature des programmes en utilisant des assertions pour spécifier les conditions de synchronisation.
2) compléter les programmes en utilisant des mécanismes de synchronisation (sémaphores ou moniteurs) qui respectent les assertions.

Le passage des assertions aux mécanismes est rendu plus difficile par des contraintes de performance ou d'implantation. Par exemple, on peut remplacer une assertion par une boucle d'attente (while) qui vérifie continuellement l'assertion et attend qu'elle soit satisfaite. Ceci n'est pas efficace, le CPU demeure continuellement occupé à vérifier l'assertion. Il s'agit d'une attente active. Il serait préférable de suspendre le processus et de passer le contrôle du CPU à un autre processus. On reprendrait le processus seulement s'il se produit un événement qui rendrait l'assertion satisfaite. Dans ce cas, qui vérifie l'assertion ? le noyau, le processus qui cause un événement qui modifie la valeur de l'assertion.

Par exemple, pour les sémaphores, l'opération P() ou wait() correspond à attendre que la valeur du sémaphore soit plus grande que zéro. Ceci peut facilement être mis en correspondance avec une assertion demandant que la valeur d'une variable soit plus grande que zéro.

Selon le problème considéré, un mécanisme de synchronisation peut s'avérer plus appropriée que les autres. Il colle mieux à la structure du problème et simplifie la programmation. On peut dire que c'est un mécanisme naturel pour ce type de problème. Même s'il simplifie la programmation, ce mécanisme peut ne pas être le plus performant. Il faut alors envisager une transformation vers un mécanisme plus performant. On reconnaît généralement un pouvoir d'expression (ou de résolution de problème) équivalent entre les principaux mécanismes. Il y a donc possibilité de passer d'un à l'autre pour des raisons de performance, même si la programmation devient plus compliquée.

La programmation implique un aspect espace (variable en mémoire) et un aspect temps (CPU). La coopération entre processus prend aussi un aspect temps (comme dans l'utilisation de sémaphore qui suspend au besoin un processus) et un aspect espace (comme dans la mémoire partagée ou la transmission de message).

1.1.4. Vérification de programme

Assurer l'exactitude (ou qu'un programme est correct) est très difficile pour les programmes concurrents. L'utilisation d'assertion entre chaque instruction peut servir d'élément de départ.

Vérification formelle et vérification automatique

La vérification formelle consiste à vérifier que l'assertion qui suit une instruction découle de la véracité de l'assertion qui précède l'instruction et des transformations produites par l'instruction. Dans la programmation concurrente, ces transformations doivent tenir compte de ce qui se passe dans les autres programmes. La vérification formelle est généralement difficile et difficilement utilisable dans des cas complexes.

Il existe une méthode par diagramme d'état qui permet de vérifier la synchronisation de programme par sémaphore. On peut détecter automatiquement la présence de blocage, c'est-à-dire la présence d'états qui peuvent conduire à un blocage. Notez que la présence d'un tel état ne produit pas nécessairement un blocage. Le blocage n'est pas automatique, par chance il peut ne pas se produire. Le programme est cependant pas fiable, les risques de blocage sont toujours présents. L'utilisation de cette méthode peut être plus difficile si l'exécution d'une opération sur un sémaphore est conditionnelle ou dépend de la valeur d'une variable.

Une autre approche consiste à considérer toutes les séquences d'instruction possibles. Toutes les séquences sont représentées dans un graphe avec des cycles (boucles). La génération de ce graphe est possible seulement pour certains mécanismes de synchronisation. Le graphe peut ensuite être examiné pour déterminer les propriétés des programmes.

Les réseaux de Pétri sont également utilisés pour modéliser la synchronisation entre plusieurs processus. Il existe des méthodes pour analyser les propriétés d'un réseau de Pétri. Cependant, les synchronisations ne peuvent pas toutes facilement être représentées par un réseau de Pétri.

Il existe donc des approches formelles mais elles demeurent cependant difficile d'application, soit parce qu'elles sont trop restrictives (ne s'appliquent que pour quelques cas), soit qu'elles sont trop complexes.

Approche opérationnelle

L'approche opérationnelle consiste à faire tourner les programmes, à observer le déroulement et à analyser le comportement. C'est une approche informelle mais qui fait aussi appel à l'analyse. L'observation conduit à l'élaboration d'un modèle de comportement. L'analyse est cependant nécessaire pour s'assurer que toutes les séquences éventuellement critiques ont été examinées et étudiées. Elle peut impliquer une approche progressive et itérative, et utilisant la méthode de essais-erreurs. A force de regarder un problème sur tous les angles, on finit par bien le comprendre et en maîtriser toutes les facettes.

1.2. Mécanisme de synchronisation ou coopération

Après avoir compris l'importance de synchroniser des processus, nous devons nous interroger sur comment le faire. Vous pouvez vous référer au livre de Silberschatz qui examine les mécanismes suivants:

- par ajout de code dans les programmes
- par ajout matériel, instruction test-and-set
- par sémaphore
- par région critique
- par moniteur

Les sémaphores impliquent seulement un appel à des fonctions du système. Les moniteurs et les régions critiques doivent être pris en compte par le langage et le compilateur. Ils offrent d'avantage de sécurité et de transparence pour la programmation, mais ils impliquent un changement plus important à l'environnement de programmation. Ils s'inscrivent dans l'orientation de la programmation objet et la syntaxe de la programmation est modifiée dans ce sens.

1.2.1. Solution logiciel à l'exclusion mutuelle (deux processus)

Nous présentons maintenant les premières tentatives pour développer un algorithme qui assure l'exclusion mutuelle. Le problème des sections critiques nous montre qu'il faut être très prudent lorsque deux programmes utilisent le même espace mémoire (les mêmes variables). La section critique est précédée de code d'*entrée* et suivi de code de *sortie* qui ont pour but d'assurer la protection de la section critique. Ce code implante un algorithme ou une manière de gérer l'accès à la section critique. Ce code utilisera généralement quelques variables partagées par les divers processus et qui pourraient agir comme des verrous. Ces variables étant partagées, le code qui les utilise (code d'entrée et de sortie) devra être écrit avec beaucoup de soins pour garantir en tout temps la consistance de leurs valeurs.

Nous rappelons que les instructions des langages de haut niveau sont généralement traduites en plusieurs instructions assembler. Les instructions assembler sont atomiques. C'est-à-dire que sur un système mono-processeur, le transfert du CPU ne peut s'effectuer qu'entre les instructions assembler.

Problème de la section critique

Un seul processus peut exécuter le code de sa section à la fois.

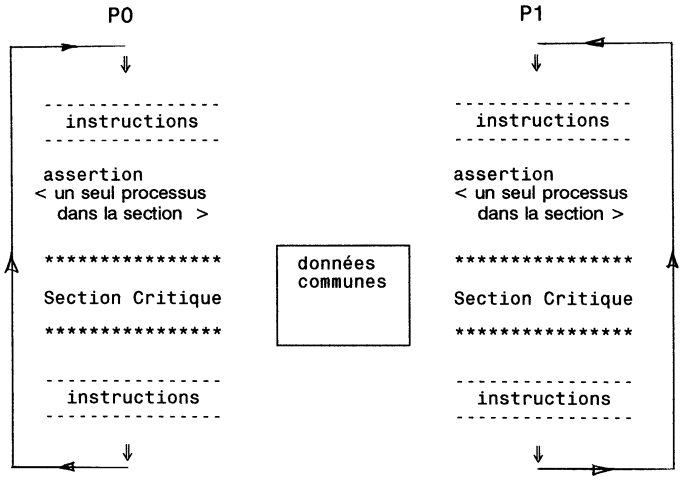


Figure 2 :Problème de la section critique

Nous allons maintenant examiner différents algorithmes pour le code d'entrée et de sortie. Nous considérons seulement le cas avec deux processus. Les premiers algorithmes présentés sont erronés et ne donnent pas toujours le résultat demandé. Seulement le dernier est correct.

La Figure 2 présente le problème de la section critique. Nous avons deux processus, P0 et P1 qui s'exécutent en concurrence. Chaque processus possède une section critique où il accède à une ressource commune, par exemple, une variable. Dans ce cas, les processus doivent éventuellement posséder des instructions permettant de déclarer et d'accéder à cette variable commune. La description d'un processus est schématique, montrant la section critique et des instructions non-déterminées qui l'entourent. Cette représentation met l'accent sur la synchronisation et non sur le but du programme lui-même. Ainsi, il n'est pas important de savoir précisément de quoi sont composées les instructions. On doit plutôt simplement considérer le fait que l'exécution de ces instructions prend du temps. De plus, on doit considérer ce temps d'exécution comme grandement variable. En effet, un bloc d'instruction peut contenir des boucles et/ou des branchements (if), rendant le temps d'exécution difficilement prévisible. De plus, dans un système multi-usager, un processus peut être interrompu à tout moment, pour passer le contrôle du CPU à un autre processus. Ceci rend donc le rythme de progression d'un programme grandement imprévisible et indéterminé. Tout ce que le programme précise est l'ordre d'exécution: nous avons quelques instructions qui doivent être exécutées avant la section critique et d'autres après. C'est dans ce contexte de rythme imprévisible que nous devons penser et concevoir la synchronisation des processus. Dans les processus P0 et P1, la section critique est incluse dans une boucle représentée par les flèches allant du bas vers le haut. La boucle est sans fin. Les processus demandent donc de multiples entrées dans la section critique. La séquence des instructions impose qu'un processus alterne entre "être dans la section critique" et "être en dehors", le temps dans chaque état étant indéterminé mais fini.

Vous pouvez référer à Silberschatz et Galvin, 4^{e} ed., pour avoir une première tentative de solution (algorithme # 1) au problème.

Algorithme # 2: (solution erronée)

La Figure 3 présente l'algorithme #2 de Silberschatz, page 141. Dans cet algorithme, le processus P0 (P1) met d'abord le *flag_0* (*flag_1*) à *vrai* pour indiquer qu'il est prêt à entrer dans sa section critique. Après cela, P0 (P1) vérifie si le processus P1 (P0) n'est pas lui aussi prêt à entrer dans sa section critique. Si P1 (P0) est prêt, alors P0 (P1) va attendre jusqu'à ce que P1 (P0) indique qu'il n'a plus besoin d'être dans sa section critique en mettant le *flag_1* (*flag_0*) à *faux*. A ce moment, P0 (P1) pourra alors entrer dans sa section critique. Lorsqu'il quitte la section critique, P0 (P1) met le *flag_0* (*flag_1*) à *faux* pour autoriser l'autre processus (s'il est en attente) à entrer dans sa section critique.

Algorithme # 2

Tentative de solution à la section critique par ajout de code à l'entrée et à la sortie

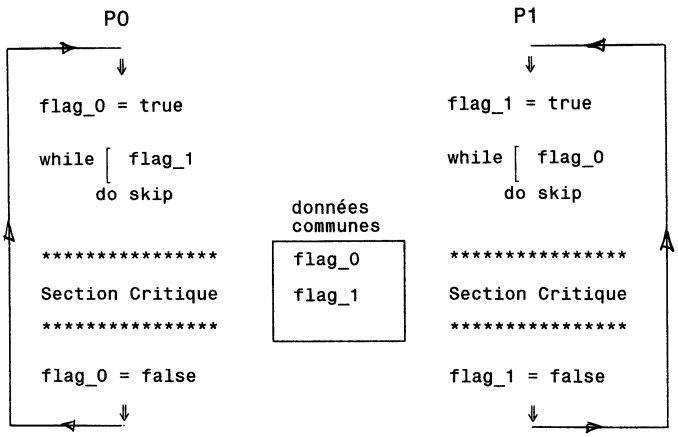


Figure 3 : Algorithme # 2

Cette solution garantit qu'un seul processus est dans la section critique à la fois. L'exclusion mutuelle est satisfaite. Malheureusement, il peut y avoir inter-blocage. Ceci implique aussi que la condition de progression de la section 1.1.2 n'est pas satisfaite. Ce cas est illustré à la Figure 5 où P0 et P1 bouclent indéfiniment dans leur instruction *while*.

On peut remarquer ici l'importance du "timing", c'est-à-dire, du temps où chaque processus exécute ses instructions ou de son rythme de progression. Ainsi, pour obtenir le blocage, il faut que le CPU soit transféré (changement de contexte) juste après l'instruction "flag_0 := true" ou "flag_1 := true".

La Figure 4 présente une version légèrement modifiée de l'algorithme #2: l'instruction "flag_0 := true" est placée après le *while*. Cette situation évite le blocage, mais elle permet éventuellement aux deux processus d'exécuter leur section critique en même temps. Donc, on ne peut garantir que l'exclusion mutuelle sera respectée.

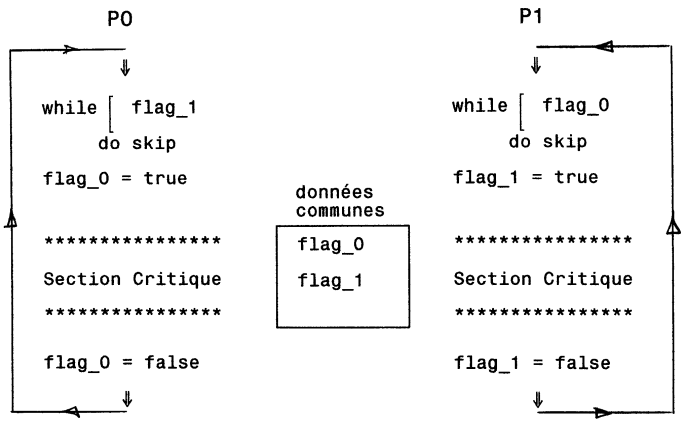


Figure 4 : Version alternative de l'algorithme # **2.**

Séquence d'exécution produisant un blocage pour l'algorithme # 2

flag_0	flag_1	P0	P1
F	F		
		.....	
F	F		
		flag_0 = true	
T	F		
			.....
T	F		
			flag_1 = true
T	T		
		while flag_1	
		do skip	
T	T		
			while flag_0
			do skip
T	T		
		while flag_1	
		do skip	
T	T		
			while flag_0
			do skip
T	T		
		while flag_1	
		do skip	
T	T		
			while flag_0
			do skip
T	T		

Figure 5 :Séquence d'exécution produisant un blocage pour l'algorithme # 2

Séquence d'exécution sans exclusion mutuelle pour la version alternative de l'algorithme # 2

flag_0	flag_1	P0	P1
F	F		
		.....	
F	F		
			.....
F	F		
		while flag_1	
F	F		
			while flag_0
F	F		
		flag_0 = true	
T	F		
			flag_1 = true
T	T		
		Section Critique	
T	T		

P0 ET P1 EXÉCUTENT EN SECTION CRITIQUE SIMULTANÉMENT

			Section Critique
T	T		
		Section Critique	
T	T		
			Section Critique
T	T		
		flag_0 = false	
F	T		
			flag_1 = false
F	F		

Figure 6 : Séquence d'exécution sans exclusion mutuelle pour la version alternative de l'algorithme # 2.

Algorithme # 3: (solution correcte)

La Figure 7 présente une solution correcte au problème d'exclusion mutuelle par ajout de code (solution logiciel). On utilise *flag_0* et *flag_1* pour indiquer l'intention des processus P0 et P1 d'entrer dans leur section critique. On utilise de plus une variable *tour* pour éviter les blocages tout en préservant l'exclusion mutuelle. Au départ, *flag_0 = flag_1 = vrai*, et la valeur de *tour* est quelconque (soit 0 ou 1).

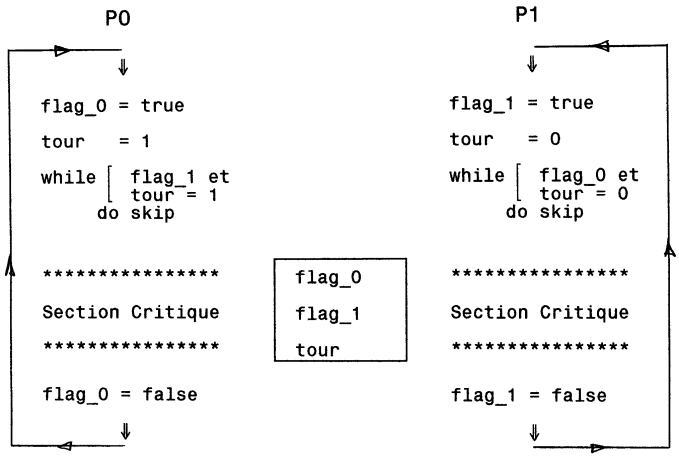


Figure 7 : Algorithme # 3. Solution correcte à la section critique proposée par Peterson, 1981.

Pour entrer dans sa section critique, le processus P0 (P1) met d'abord *flag_0* (*flag_1*) à *vrai*, et alors suppose que c'est au tour de l'autre processus d'entrer dans sa section critique en mettant *tour* à 1 (0). Si les deux processus essaient d'entrer en même temps, les deux valeurs 0 et 1 seront assignées à *tour* presque en même temps. L'opération d'assignation (de copie) d'une valeur à une variable (position mémoire) peut cependant être considérée comme une opération atomique. Donc, des deux valeurs assignées à *tour*, une seule va durer. La première valeur sera rapidement remplacée par la seconde. La dernière valeur de *tour* déterminera lequel des deux processus peut entrer dans sa section critique. Cependant, peut importe si la première ou la seconde valeur est 0 ou 1, l'algorithme assurera toujours l'exclusion mutuelle. La page suivante présentent 4 séquences d'exécution illustrant le fonctionnement de l'algorithme.

Pour prouver que cette solution est correcte, nous devons démontrer 1) que l'exclusion mutuelle est préservée, 2) qu'il n'y a pas d'inter-blocage, et 3) qu'il n'y a pas de famine.

Pour démontrer l'exclusion mutuelle, nous notons que le processus P0 entre dans sa section critique seulement si soit *flag_1 = faux* ou soit *tour* = 0. De plus, pour que les deux processus exécutent leur section critique en même temps, il faudrait que *flag_0 = flag_1 = vrai*. Ces deux remarques impliquent que P0 et P1 ne peuvent pas avoir réussi (être sorti de la boucle) l'instruction *while* en même temps, parce que la valeur de *tour* est soit 0 ou 1 mais pas les deux. Supposons que P0 est en train d'exécuter le test de la boucle *while*, et examinons les cas possibles selon la progression de P1.

1) Si P1 n'a pas encore exécuté l'instruction "flag_1 := true", alors le test de P0 est faux (car *flag_1 = faux*) et P0 entre dans la section critique.
2) Si P1 a exécuté l'instruction "flag_1 := true" mais pas encore "tour := 0", alors le test est vrai et P0 va boucler dans le *while* jusqu'à ce que P1 exécute l'instruction "tour := 0". Alors, P0 pourra entrer dans sa section critique tandis que P1 devra attendre dans la boucle *while*.
3) Si P1 est aussi rendu à la boucle *while*, alors soit P0 ou P1 entrera dans la section critique selon que c'est l'instruction "tour := 0" de P1 ou l'instruction "tour := 1" de P0 qui a été exécutée en dernier.

Notons que le seul endroit où un processus peut être bloqué est dans la boucle *while*. Il ne peut donc pas y avoir d'interblocage (c'est-à-dire que les deux processus se bloquent mutuellement) car les deux ne peuvent être bloqué simultanément dans la boucle *while*. En effet, selon la valeur de *tour*, un processus sera forcément libéré.

Les possibilités de famine sont également évitées. Supposons que P0 est en attente dans la boucle *while* tandis que P1 utilise la section critique. Lorsque P1 quittera la section critique, il mettra *flag_1* à *faux* et permettra ainsi à P0 de continuer. P1 pourrait cependant essayer d'entrer à nouveau dans sa section critique avant que P0 ait l'occasion de le faire. Pour cela, P1 devra exécuter "flag_1 := true" et "tour := 0". La première instruction va bloquer P0 dans la boucle *while* mais la seconde va aussitôt le libérer. Pendant ce temps, P1 devra attendre dans la boucle *while* car *tour* = 0. Donc, P0 devra attendre après au plus une entrée de P1 pour pouvoir entrer à son tour dans la section critique. Donc, il n'y a pas de famine.

Séquence d'exécution possible pour l'algorithme # 3 (seq. A)

flag_0	flag_1	tour	P0	P1
F	F	X		
F	F	X	.....	
T	F	X	flag_0 = true	
T	F	1	tour = 1	
T	F	1	while flag_1 & tour=1	
T	F	1	Section Critique	
T	F	1	Section Critique	
F	F	1	flag_0 = false	
F	F	1	.....	
F	F	1		.....
F	T	1		flag_1 = true
F	T	0		tour = 0
F	T	0		while flag_0 & tour=0
F	T	0		Section Critique
F	T	0		Section Critique
F	F	0		flag_1 = false
F	F	0		.....

Séquence d'exécution possible pour l'algorithme # 3 (séq. B)

flag_0	flag_1	tour	P0	P1
F	F	X		
F	F	X	.....	
T	F	X	flag_0 = true	
T	F	1	tour = 1	
T	F	1	while flag_1 & tour=1	
T	F	1	Section Critique	
T	F	1		.....
T	T	1		flag_1 = true
T	T	0		tour = 0
T	T	0		while flag_0 & tour=0 do skip
T	T	0		while flag_0 & tour=0 do skip
T	T	0	Section Critique	
T	T	0	flag_0 = false	
F	T	0	.....	
F	T	0		while flag_0 & tour=0
F	T	0		Section Critique
F	T	0		Section Critique
F	T	0		flag_1 = false
F	F	0		.....
F	F	0		

Séquence d'exécution possible pour l'algorithme # 3 (séq. C)

flag_0	flag_1	tour	P0	P1
F	F	X		
F	F	X	.....	
T	F	X	flag_0 = true	
T	F	1	tour = 1	
T	F	1		.....
T	T	1		flag_1 = true
T	T	0		tour = 0
T	T	0	while flag_1 & tour=1	
T	T	0		while flag_0 & tour=0 do skip
T	T	0		while flag_0 & tour=0 do skip
T	T	0	Section Critique	
T	T	0	Section Critique	
T	T	0		while flag_0 & tour=0 do skip
T	T	0	flag_0 = false	
F	T	0		while flag_0 & tour=0
F	T	0		Section Critique
F	T	0		Section Critique
F	T	0		flag_1 = false
F	F	0		.....
F	F	0		

Séquence d'exécution possible pour l'algorithme # 3 (séq. D)

flag_0	flag_1	tour	P0	P1
F	F	X		
F	F	X	.....	
T	F	X	flag_0 = true	
T	F	X		.....
T	T	X		flag_1 = true
T	T	0		tour = 0
T	T	1	tour = 1	
T	T	1	while flag_1 & tour=1 do skip	
T	T	1		while flag_0 & tour=0
T	T	1		Section Critique
T	T	1	while flag_1 & tour=1 do skip	
T	T	1		Section Critique
T	T	1	while flag_1 & tour=1 do skip	
T	T	1		flag_1 = false
T	F	1	while flag_1 & tour=1	
T	F	1	Section Critique	
T	F	1	Section Critique	
T	F	1	flag_0 = false	
F	F	1		.....
F	F	1		

1.2.2. Solution matérielle à la section critique

La solution au problème de la section critique implique l'utilisation d'opération atomique. Le problème serait facilement résolu si nous pouvons empêcher les interruptions de se produire lorsque nous modifions la valeur d'une variable partagée (pour la multi-programmation sur un seul processeur). Malheureusement, cette solution n'est pas toujours réalisable. Plusieurs machines fournissent cependant des instructions assembler (matérielles) spéciales qui permettent soit 1) de tester et modifier le contenu d'un mot mémoire, ou 2) d'interchanger (swap) le contenu de deux mots, et cela de manière atomique. Ces instructions spéciales permettent de résoudre le problème des sections critiques de manière relativement simple.

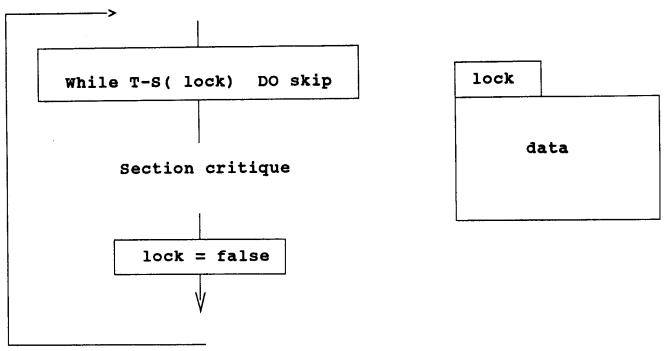
L'instruction **Test-and-Set** est défini par:

```
function  Test-and-Set ( var lock : boolean )
                  : boolean
    begin
        var  old_value  := lock ;
        lock            := true ;
        return  old_value;
    end ;
```

L'instruction retourne la valeur actuelle de la variable (lock) et en même temps met cette valeur à *vrai*. L'important est que cette instruction est exécutée de manière atomique. Ainsi, si deux instructions **Test-and-Set** sont exécutées simultanément (chacune sur un CPU différent), elles seront en fait exécutées séquentiellement selon un ordre arbitraire.

Avec l'instruction **Test-and-Set**, nous pouvons implanter l'exclusion mutuelle en déclarant une variable, *lock*, qui agit comme un verrou associé aux données communes utilisées par les sections critiques. La page suivante illustre son utilisation. Au départ, la variable *lock* est initialisée à *faux*, indiquant qu'aucun processus n'a réservé l'utilisation des données. Lorsqu'un processus veut utiliser les données, il doit attendre que la valeur de *lock* soit *faux*, et en même temps, mettre *lock* à *vrai* pour empêcher les autres processus d'utiliser les données. Cette solution comporte cependant des possibilités de famine. Une solution sans risque de famine est présentée par Silberschatz page 150.

TEST-AND-SET



1.2.3. Sémaphores

SÉMAPHORE

mécanisme de synchonisation supporté par le noyau

définition de DIJKSTRA

Type abstrait - associé à une valeur entière

initialisation --> Semaphore S = 1;

wait (S) **P (S) - test**

```
while S ≤ 0 do no-op
S = S - 1
```

signal (S) **V (S) - incrémente**

```
S = S + 1
```

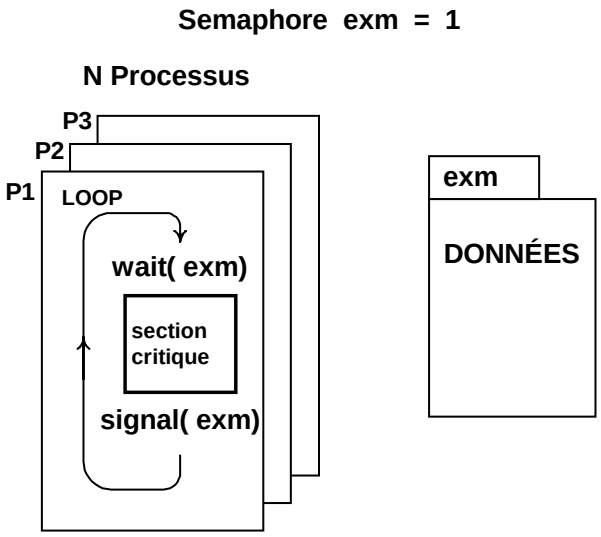
Les solutions au problème de la section critique présentées jusqu'ici sont difficiles à généraliser pour des problèmes plus complexes. Nous présentons maintenant un mécanisme de synchronisation plus intéressant: les sémaphores. Un sémaphore S est une variable entière qui est accédée par seulement deux opérations standards et *atomiques*: P() et V(). Selon le langage de programmation, il existe également une manière de déclarer et d'initialiser les sémaphores. Les opérations P et V sont aussi appelées *wait* pour P et *signal* pour V. Les termes *wait* et *signal* sont sans doute plus significatifs, mais ils sont également utilisés dans les moniteurs que nous examinerons bientôt. Aussi, pour éviter la confusion, nous utiliserons la notation P-wait() et V-signal() pour les sémaphores. Les définitions classiques et standards de P-wait et V-signal sont:

```
P-wait( S ) :    while S <= 0  do  no-op ;
                 S := S - 1 ;

V-signal( S ) :  S := S + 1 ;
```

La modification de la valeur entière du sémaphore dans P-wait et V-signal est effectuée de manière indivisible. Donc, lorsqu'un processus modifie la valeur du sémaphore, aucun autre processus ne peut simultanément modifier le même sémaphore. De plus, dans l'opération P-wait, Le test de la valeur (l'instruction while) et la modification de la valeur (l'instruction "S := S - 1" doivent s'exécuter sans interruption, d'une manière atomique.

L'implantation des sémaphores est généralement différente de la définition standard. La définition indique le comportement des opérations P-wait et V-signal, et toute implantation doit suivre ce comportement. Normalement, on doit utiliser seulement les opérations P-wait et V-signal avec les sémaphores. Un programme développé à partir de la définition devrait fonctionner correctement sur n'importe quelle implantation. Notons que, selon la définition, la valeur du sémaphore n'est jamais négative. Une implantation peut cependant faire que la valeur devienne négative. C'est pourquoi, il est interdit d'examiner et/ou de modifier la valeur d'un sémaphore. Normalement, un sémaphore est initialisé à 0. Certain langage ou environnement permette de spécifier (modifier) la valeur d'initialisation.

La figure suivante indique comment un sémaphore peut être utilisé pour résoudre le problème de la section critique avec N processus. Le sémaphore exm peut être considéré comme un verrou qui assure qu'un seul processus accède aux données à la fois (i.e., exécute sa section critique). Initialement, la valeur de exm est 1 et aucun processus ne s'exécute dans sa section critique. Le premier processus qui effectue l'opération P-wait(exm) pourra donc accéder aux données et mettra la valeur de exm à 0. Tout autre processus effectuant l'opération P-wait(exm) sera donc mis en attente (sera bloqué). Lorsque le premier processus quitte la section critique, il effectue l'opération V-signal(exm). V-signal(exm) incrémente le sémaphore (exm = 1), ce qui permet de débloquer un processus en attente et un seul.



Le sémaphore est très utile pour résoudre des problèmes de synchronisation simple. Par exemple, considérons deux processus, P1 et P2, où P1 est composé des instructions Ins1 et Ins2 et P2 est composé de Ins3 et Ins4. Si nous voulons que l'instruction Ins3 soit exécuté seulement après l'instruction Ins1, alors nous pouvons utiliser un sémaphore *synch* comme suit:

P1
```
Ins1 ;
V-signal(synch) ;
Ins2 ;
```

P2
```
P-wait(synch) ;
Ins3 ;
Ins4 ;
```

le sémaphore *synch* est initialement égal à 0. P2 sera donc bloqué sur P-wait tant que P1 n'aura pas effectué l'instruction "V-signal(synch)". Les sémaphores sont donc bien utiles pour imposer une séquence d'exécution particulière aux instructions de plusieurs processus.

Attente active

La définition du sémaphore utilise une attente active dans l'opération P-wait: la boucle *while*. L'opération P-wait indique que tant que S <= 0, le processus attend. L'utilisation d'une boucle *while* définit bien cette attente. Il nous faut cependant faire la distinction entre la définition et l'implantation. Une attente active n'est pas acceptable au niveau de l'implantation. Ceci est un problème sérieux pour la multi-programmation sur un système avec un seul processeur. Pendant qu'un processus exécute une attente active (par exemple, est en attente dans un P-wait) les autres processus ne peuvent pas avancer.

IMPORTANT: dans tout problème de synchronisation que je peux vous demander, il est strictement interdit d'utiliser l'attente active.

SÉMAPHORE

Implantation sans attente active
sans "busy - waiting"

initialisation --> Semaphore S (1);

```
class Semaphore
      int      count;  // valeur du sémaphore
      list_t   list;   // liste de processus
```

wait ( ) **S.wait ( ) ;**

```
count = count - 1 ;
if ( count < 0 )
{   ajoute ce processus à list ;
    bloque ce processus
}
```

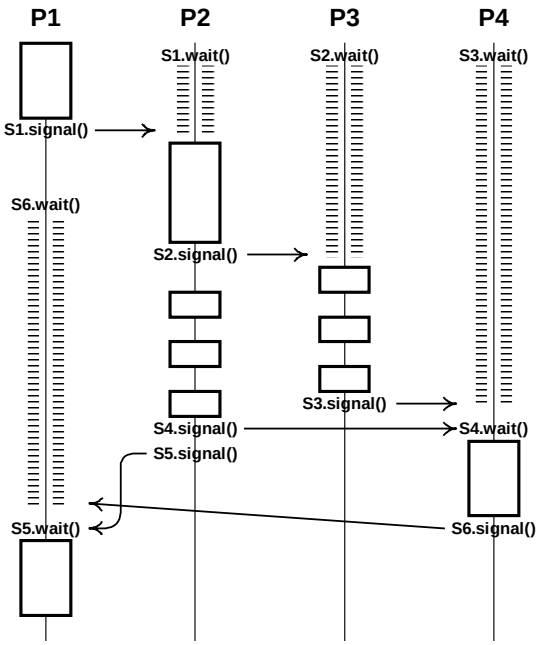
signal ( ) **S.signal ( ) ;**

```
count = count + 1 ;
if ( count ≤ 0 )
{   enlève un processus P de list;
    réveille le processus P;
}
```

Une implantation des sémaphores doit éviter l'attente active. Pour cela, on peut demander au système d'exploitation (au noyau) de bloquer (suspendre) le processus. Ceci implique que le noyau place le processus dans une queue d'attente et passe le CPU à un autre processus non en attente. L'opération P-wait devra donc faire un appel au noyau pour lui demander de suspendre le processus courant, tandis que l'opération V-signal demandera au noyau de réveiller un des processus en attente. Cette implantation est présentée dans la figure précédante. Le sémaphore est alors associé à une structure possédant deux éléments: un entier pour la valeur du sémaphore et une liste de processus contenant tout les processus en attente sur ce sémaphore. Notez que l'opération P-wait

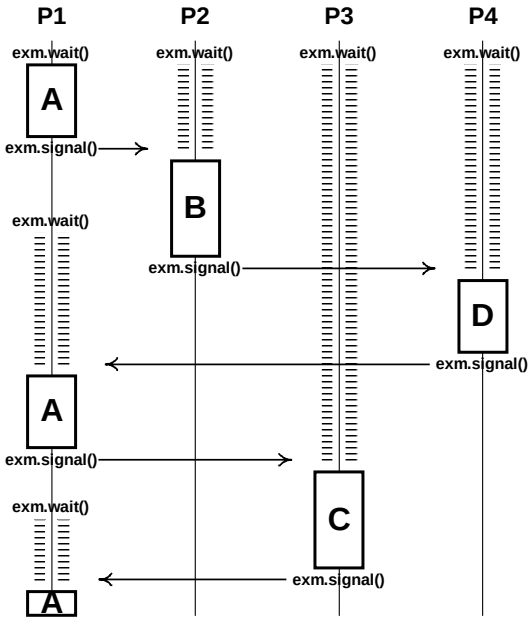
décrémente la valeur du sémaphore avant de vérifier si le processus doit être suspendu. Le sémaphore peut donc avoir une valeur négative, et cette valeur indiquera combien de processus sont en attente. Une opération V-signal retire un processus de la queue (s'il y en a un). Il n'est pas spécifié dans quel ordre les processus sont retirés de la queue. Différentes gestions de liste peuvent être utilisées. Il faut cependant utiliser un mécanisme qui soit équitable envers tout les processus. Il faut éviter les risques de famine (attente infini). On peut, par exemple, utiliser une queue FIFO, premier entré - premier sorti. Le développement de programme ne doit pas prendre en compte le mécanisme de gestion de queue utilisé. Le programme devrait fonctionner correctement pour tout gestion de queue équitable.

CONTRÔLE DE SÉQUENCE OU D'ORDRE D'EXÉCUTION PAR SÉMAPHORES



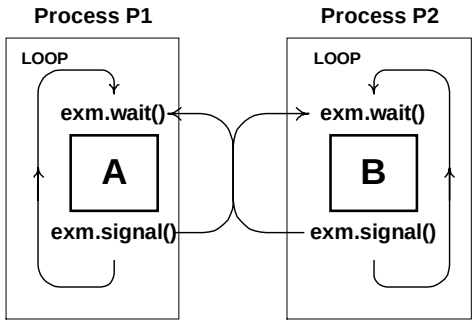
EXCLUSION MUTUELLE

Semaphore exm = 1



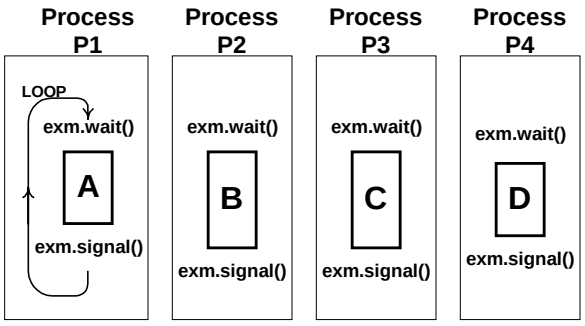
EXCLUSION MUTUELLE

Semaphore exm = 1



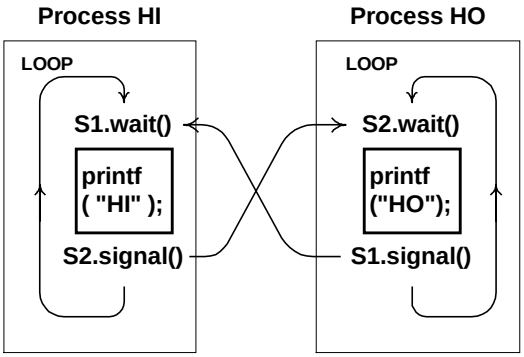
EXCLUSION MUTUELLE

Semaphore exm = 1



ALTERNANCE FORCÉE

Semaphore S1 = 1
Semaphore S2 = 0



1.2.4. Langage concurrent

L'écriture de programme concurrent peut être simplifiée si nous possédons un langage avec des instructions (ayant au besoin une syntaxe propre) dédiées à la synchronisation de processus. Le langage Pascal a été développé dans le but de faciliter l'écriture de programme correct. Le langage encourage une programmation structurée. L'utilisation de mécanisme d'abstraction comme la définition de type par l'usager permet d'améliorer la clarté d'un programme. L'utilisation de région critique et de moniteur dans la programmation concurrente permet souvent de rendre le programme plus clair et compréhensible. Ces mécanismes s'inscrivent dans l'esprit de la programmation structurée et de la programmation par objet.

On peut ajouter à un langage de programmation la notion de sémaphore sans avoir à modifier le langage ou à introduire de nouvelle instruction. Par exemple, dans le système Unix, le système d'exploitation fournit des fonctions accomplissant les opérations P-wait et V-signal. P-wait et V-signal sont des fonctions que tout programme peut appeler (peut importe le langage). Dans Unix, les sémaphores sont gérés par le noyau et sont créés par une opération comparable à une ouverture de fichier. L'opération de création retourne un identificateur (nombre entier) qui doit par la suite être utilisé avec les opérations P-wait et V-signal pour indiquer sur quel sémaphore il faut effectuer l'opération. Il n'y a donc pas de type sémaphore et de déclaration de sémaphore. L'utilisation des sémaphores est un peu obscurcie par le fait que l'on doit passer par l'intermédiaire d'un identificateur.

Un langage concurrent avec le type sémaphore est cependant possible. La déclaration d'un sémaphore serait de la forme suivante:

```
var  mutex  :  semaphore ;
var  mutex  :  semaphore := 1 ;
```

Ces instructions déclarent *mutex* comme un sémaphore. Dans le second cas, on initialise également le sémaphore à 1. *mutex* est ici vraiment une variable définissant un sémaphore. Les deux seules opérations possibles avec un sémaphore sont P-wait et V-signal.

Un langage concurrent possède aussi généralement une instruction *process* qui permet de définir des processus qui s'exécuteront en concurrence. L'instruction est similaire à la définition de fonction ou procédure. La différence est qu'un *process* correspond en fait à un programme principal et s'exécute donc sans être appelé. De plus, s'il y a plusieurs *process*, ils s'exécutent en parallèle. (Certain langage demande qu'un *process* soit appelé pour commencer son exécution.)

1.2.5. Région critique

Avec les sémaphores, il est souvent difficile de s'assurer que le programme est correct. Pour un problème moyennement complexe, on peut facilement commettre une erreur. Avec une erreur, le programme peut quand même dans beaucoup de cas produire le bon résultat. Cependant, le programme dans son ensemble est incorrect: on ne peut garantir que le programme va toujours donner le bon résultat. Un programme incorrect peut donc être difficile à reconnaître. Avec les sémaphores, une seule erreur d'inattention peut rendre le programme incorrect.

Les régions critiques impliquent l'utilisation d'un type nommé *shared* et d'une instruction nommée *region*. La page suivante présente l'utilisation de région pour contrôler l'accès à une section critique. La variable *data* qui est une variable de nature structure de donnée (record ... end) est défini de type *shared*. Ceci signifie que la variable peut être utilisée par plusieurs processus (mais pas en même temps). L'instruction "region data do «section critique»;" permet de spécifier la partie du code (section critique) qui utilise la variable partagée. *region* est un mot clé. *data* définit les données communes utilisées par la section critique. L'instruction *region* permet donc une programmation plus structurée. La section critique doit être définie comme un bloc d'instruction à l'intérieur de l'instruction *region*. Une erreur de l'usager sera alors détectée par le compilateur. L'instruction *region* assure qu'un seul processus exécute la section critique à la fois. L'implantation d'une région peut être fait par l'ajout d'un sémaphore qui agit comme un verrou sur les données. L'instruction *region* sera remplacée par l'ajout d'une opération P-wait avant la section critique et d'une opération V-signal après (voir figure).

La région critique est cependant un mécanisme limité pour la programmation concurrente. Pour obtenir un mécanisme plus souple, il faut utiliser les régions critiques conditionnelles (voir Silberschatz, page 163).

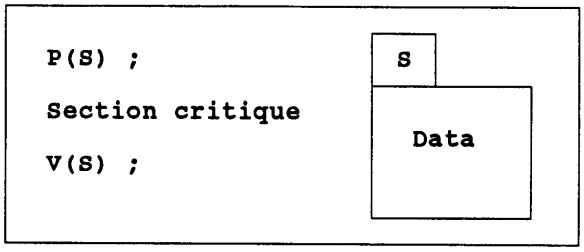
PROGRAMMATION CONCURRENTE

Introduction d'éléments de langage pour faciliter la programmation de processus concurrent et de leur synchronisation

Modurarisation

Type abstrait

sémaphore:



REGION

```
Var Data : Shared record
                        description
                        des données
                   end ;

Region Data
     DO begin
          Section critique
        end ;
```

1.2.6. Moniteur

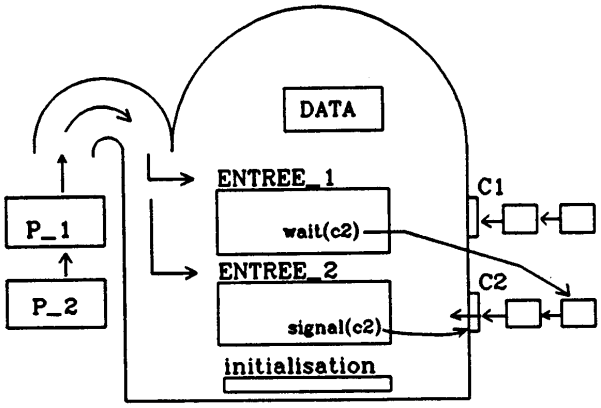
MONITEUR

Remplace "module" ou "classe" par "moniteur"

Combine ✯ module → encapsulation
✯ région critique → exclsion mutuelle

Un seul processus à la fois exécute dans un moniteur

Queue d'attente de processus



Un moniteur est une amélioration du principe de région critique. Il est également inspiré par la notion de programmation par objet. Un objet est un mécanisme d'abstraction. Un objet définit un ensemble de données (variables) ainsi qu'un ensemble de procédures qui opèrent sur les données. La définition d'un module est similaire à celle d'objet. Pour l'approche objet, il faut d'abord définir le type de l'objet. Donc la définition de l'objet est la définition de son type. Par la suite, dans un programme, on peut définir plusieurs variables ayant ce même type. Par contre, la définition d'un module n'est pas une définition de type. Il n'y a qu'une seule instance d'un module. Selon le langage, un moniteur est soit un type objet soit un module. Dans Turing Plus, un moniteur est un module, et se définit comme suit:

```
monitor mon_name
        export ....

        Déclaration des variables
        initialisation

        procedure no 1
        procedure ...
end mon_name
```

A l'intérieur du moniteur, on déclare donc un ensemble de variables qui seront utilisées par plusieurs processus. A la différence des régions critiques, le code qui utilise les variables communes (les sections critiques) n'est pas contenu dans les processus, mais fait partie du moniteur. Ce sont les procédures du moniteur. Les processus (programmes) doivent donc faire appel aux procédures du moniteur pour utiliser les variables communes. Les procédures sont les sections critiques et le moniteur assure l'exclusion mutuelle dans les procédures. C'est-à-dire qu'un seul processus à la fois peut exécuter une procédure du moniteur.

Un moniteur agit comme un gardien surveillant l'accès aux procédures. Le moniteur a la possibilité de suspendre un processus. Il dispose donc d'une queue d'attente où il place les processus en attente. Les processus doivent faire la queue pour accéder aux procédures du moniteur.

Conditions

Les moniteurs possèdent un type *condition*. Les variables de ce type ne peuvent être déclarées que dans un moniteur. Une condition est similaire à un sémaphore. Il y a deux opérations permises avec les conditions: l'opération *wait* qui suspend le processus et l'opération *signal* qui réveille un processus. Nous pouvons donc avoir les instructions :

```
(1)   var  x, y : condition
(2)   wait ( x )
(3)   signal ( y )
```

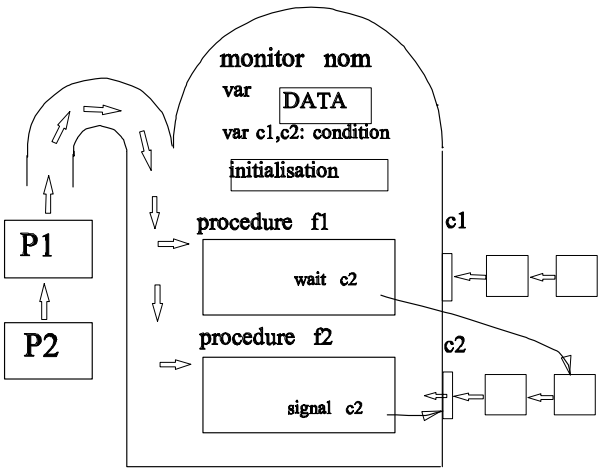
CONDITION → type abstrait

Synchronisation entre (à l'intérieur de) les procédures d'un moniteur

wait(x) → met un processus en attente dans une queue associée à la condition

signal(x) → active (réveille) un processus dans la queue "x", s'il y en a un

Question: quel processus continue après un signal ?



L'instruction (1) se place dans la déclaration des variables partagées d'un moniteur. Elle définit x et y comme étant deux conditions. A chaque condition est associée une queue d'attente.

L'opération *wait* suspend le processus (qui exécute la procédure du moniteur contenant le *wait*) sur la queue d'attente correspondant à la condition. Remarquez que le *wait* du moniteur suspend toujours le processus, ce qui est différent du P-wait des sémaphores où la suspension à lieu seulement si la valeur du sémaphore est <= 0. Une condition n'a pas de valeur. Une condition a seulement une queue d'attente.

L'opération *signal* réveille un processus suspendu sur la queue correspondant à la condition. S'il n'y a pas de processus suspendu, alors l'opération signal n'a aucun effet. Ce comportement est différent du V-signal des sémaphores qui a toujours un effet. Il réveille un processus s'il y en a un en attente, si non il incrémente la valeur du sémaphore. Le sémaphore mémorise donc le nombre de V-signal, ce n'est pas le cas du *signal* des moniteurs.

Quand un processus est suspendu sur une condition, il libère (quitte) alors le moniteur et un autre processus peut ensuite accéder à une procédure du moniteur. Lorsque qu'un processus P1 exécutant dans le moniteur réveille un autre processus P2 par une instruction *signal*, il y a alors deux processus qui pourraient s'exécuter dans le moniteur. Il est cependant important qu'un seul des processus puisse poursuivre son exécution pour assurer l'exclusion mutuelle. Il y a alors deux choix possible conduisant à deux implantations possibles des moniteurs.

1) le processus qui a effectué l'opération *signal*, P1, continue son exécution.
   (P1 était déjà en train de s'exécuter, il est plus simple de le laisser continuer.)

2) le processus réveillé, P2, a accès au moniteur et continue son exécution. P1 pourra continuer son exécution quand P2 aura quitté le moniteur.
   (P2 a été réveillé par P1 quand certaines conditions étaient satisfaites, si on attend trop longtemps pour réveiller P2, il se peut que ces conditions ne soient plus satisfaites. Du point de vue de la vérification de l'exactitude d'un programme, il est préférable de réveiller le processus immédiatement après l'instruction *signal*.)

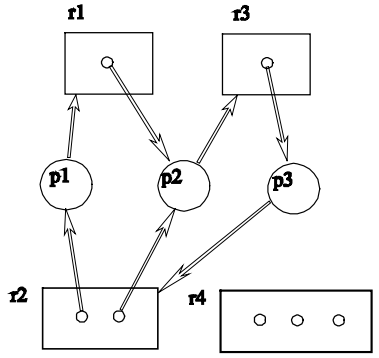
Turing Plus utilise la seconde alternative: le processus signalé (réveillé) est exécuté en premier. Nous pouvons utiliser une "condition deferrred" si nous désirons la première alternative: le processus qui effectue le signal continue son exécution. Nous disposons également de conditions avec priorité:

```
var c : condition priority
wait ( c ,  13 )
signal ( c )
```

L'instruction *wait* comporte un paramètre définissant la priorité. L'opération signal réveille le processus suspendu ayant la plus petite valeur pour le paramètre de priorité.

1.3. Interblocage et famine

Des programmes corrects impliquent qu'il n'y ait pas de "possibilité" d'interblocage ou de famine. L'interblocage implique que plusieurs processus se nuisent mutuellement, tandis que la famine implique qu'un seul processus est lésé par l'action des autres.



Contraintes structurelles - performance

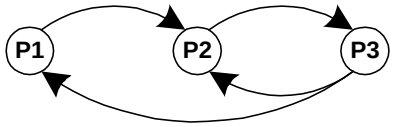
CONDITIONS d'Interblocage

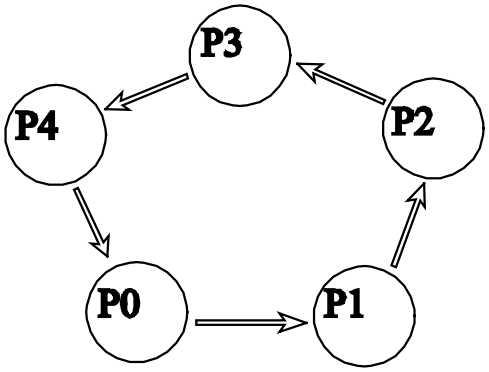
1- Exclusion mutuelle
   au moins une ressource n'est pas partageable

2- Retient et attend
   un processus retient des ressources et attend d'autres déjà allouées

3- Pas de "préemption" (pas d'expulsion)

4- Attente circulaire





ensemble circulaire
de processus en attente,
P0 attend après P1, ...

graphe d'allocation des ressources avec interblocage

INTERBLOCAGES
(Deadlocks)

Définition:

chaque processus d'un ensemble **attend** un événement qui ne peut être produit que par **un autre** processus de l'ensemble.

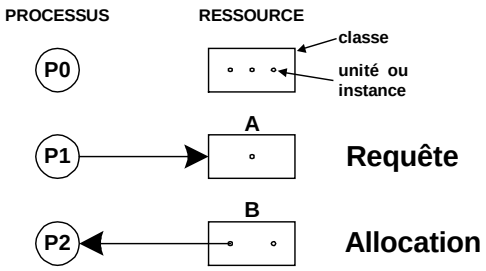
Ne pas confondre avec BLOCAGE :

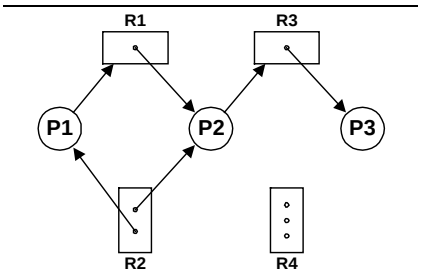
- arrêt d'un processus pour un temps indéterminé

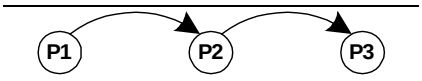
Chapitre 6 de Stalling

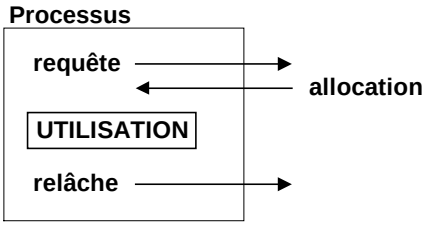
- ⇨ Graphe d'allocation des ressources
- ⇨ Conditions nécessaires
- ⇨ Prévention
- ⇨ Détection
- ⇨ Évitements

Graphe d'allocation des ressources









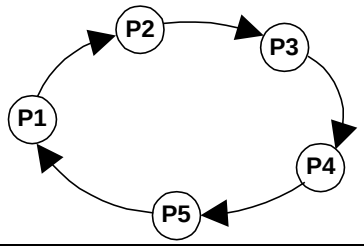
CONDITIONS D'INTERBLOCAGE

(structure) **(⇨ performance)**

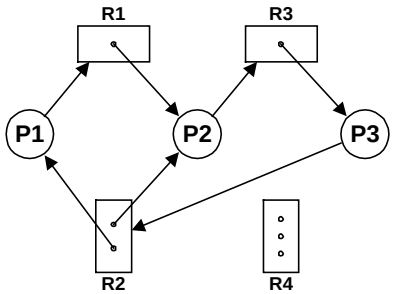
- ⇨ **Exclusion mutuelle**
  au moins une ressource n'est pas partageable
- ⇨ **Pas de "préemption" (réquisition)**

(comportement)

- ⇨ **Retient et attend**
  un processus retient des ressources et attend d'autres déjà allouées
- ⇨ **Attente circulaire**
  ensemble circulaire de processus en attente



Les 4 conditions sont NÉCESSAIRES mais NON SUFFISANTES



1.4. Formes et moyens de coopération

Forme

L'interaction et la coopération entre processus peut prendre différentes formes:

a) échange d'informations

b) synchronisation

c) partage de ressources en exclusion mutuelle

Moyens

Différents mécanismes de synchronisation et de coopération sont disponibles:

a) messages

b) sémaphores

c) région critique

d) moniteur

e) rendez-vous (Ada)

f) signal, fichiers et pipes

Ces mécanismes nécessitent selon le cas:
- l'addition de fonctions (une librairie)
- extension du langage

On peut donc se demander pour un problème donné quel mécanisme de synchronisation est préférable. Un mécanisme peut permettre une formulation plus appropriée du problème, et permettre une programmation plus facile et limpide.

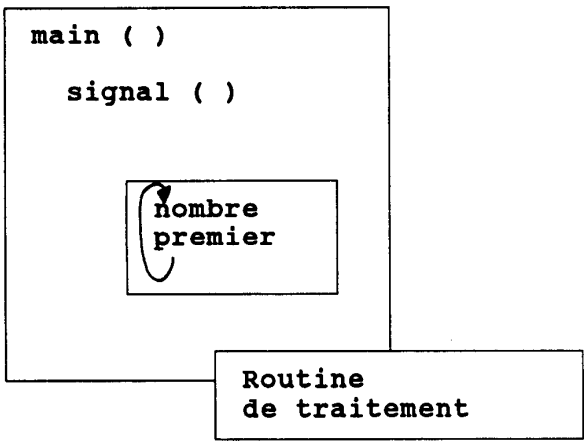
Cependant, ce mécanisme peut ne pas être le meilleur choix du point de vu de la performance. Il faudrait alors considérer la conversion du programme d'un mécanisme à un autre.

Il est reconnu que les mécanismes message, sémaphore et moniteur ont un pouvoir d'expression équivalent. C'est-à-dire que l'on peut passer d'une forme à l'autre. Tout problème qui peut être résolu par un des mécanismes peut également l'être par les autres.

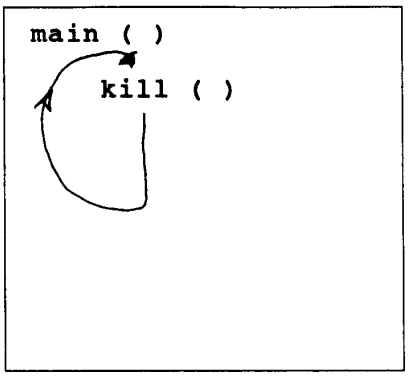
SIGNAL

- **le programme n'a pas besoin de vérifier si un signal est envoyé**

p_calcul.c



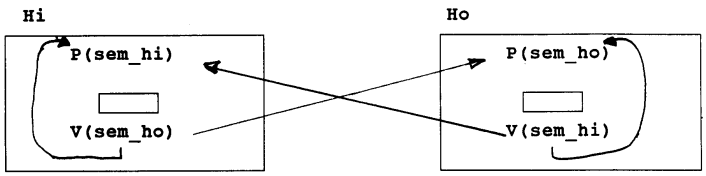
p_control.c



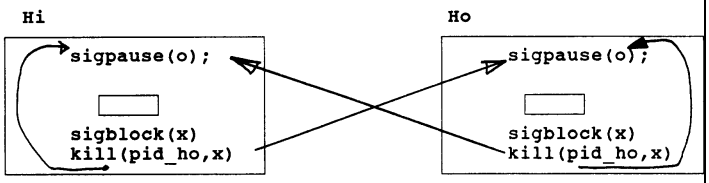
- **synchronisation → utiliser le "pause"**
- **signal est un moyen limité d'intraction**

ALTERNANCE FORCÉE (HI - Ho)

avec Sémaphore



avec signaux



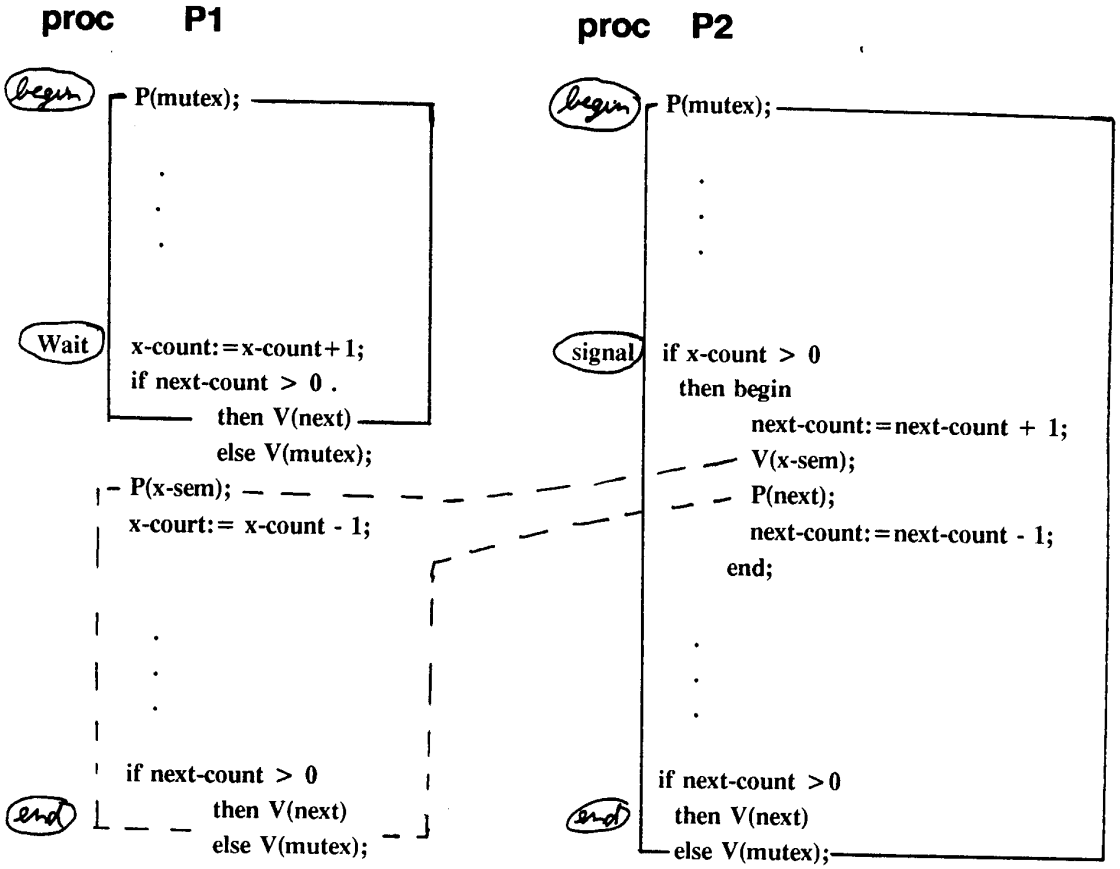
Problème → difficulté de prévoir tous les états possibles de processus concurrents

Construction de Sémaphores à l'aide d'un moniteur

```
type semaphore = monitor
     var busy: boolean;
         nonbusy: condition;

procedure entry P;
     begin
          if busy then nonbysy.wait;
          busy := true;
     end;

procedure entry V;
     begin
          busy := false;
          nonbusy.signal;
     end;

begin
     busy := false;
end.
```

Construction d'un Moniteur à l'aide de Sémaphores

Sémaphores mutex = 1, next = 0, x-sem = 0;
int next-count = 1, x-count = 0 ;



LANGAGES CONCURRENTS

SEMAPHORE -▶ fonctions systèmes

PASCAL concurrent

processus - classe - moniteur

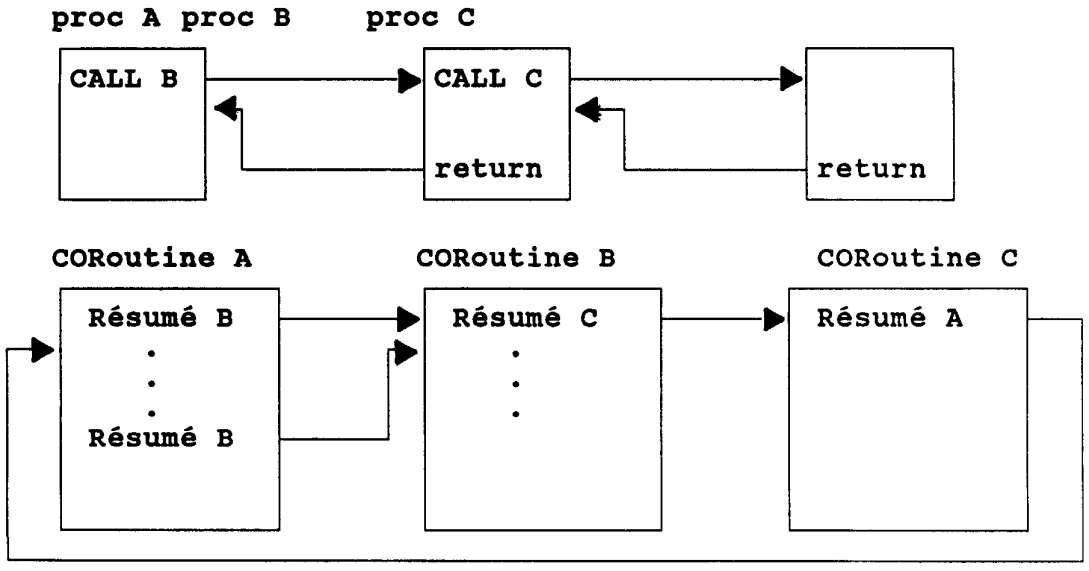
CSP - Communicating Sequential Processes
langage de spécification
transmission de messages

ADA

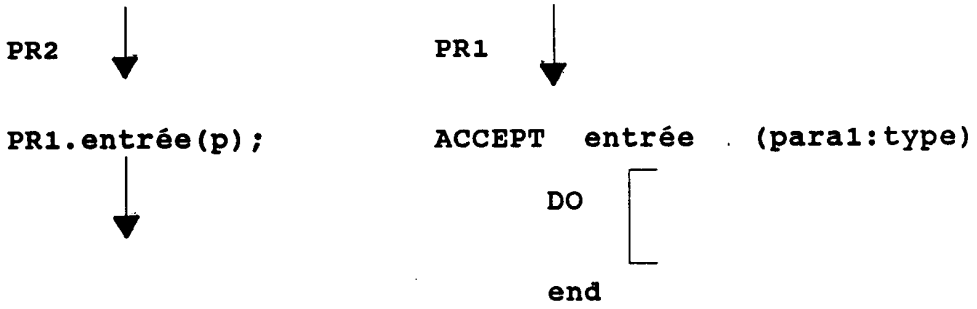
COROUTINE - modula -2

intermédiaire ▶ appel de procédure

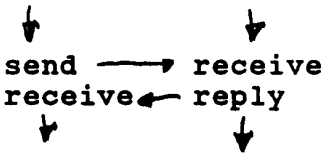
entre ▶ concurrent



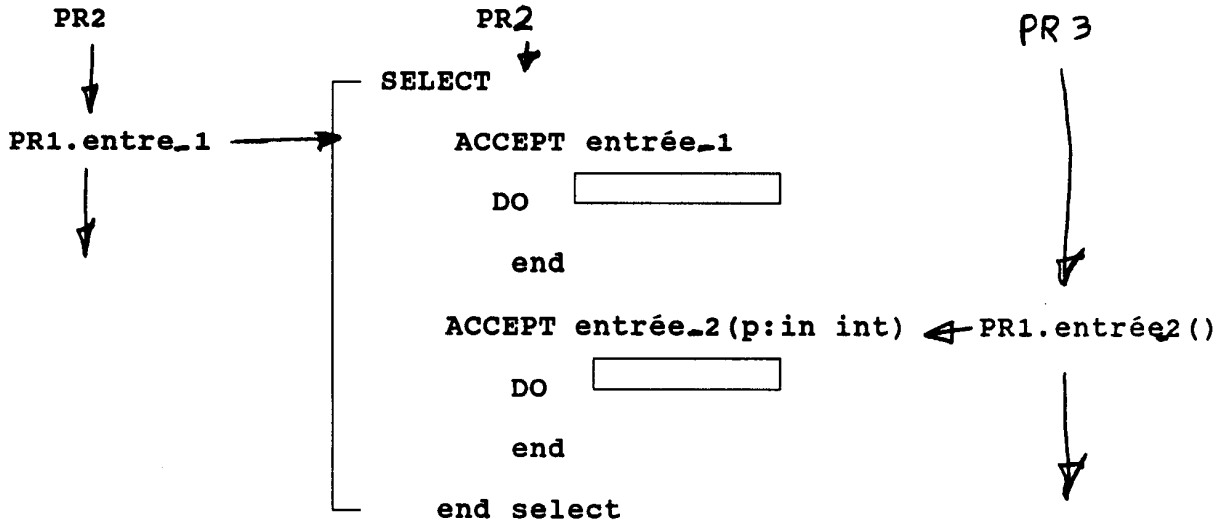
RENDEZ-VOUS dans ADA



ACCEPT → défini un point d'entrée
→ passage de paramètres (in, out)
→ rendez-vous



SELECT → choix entre plusieurs ACCEPTS



1.5. Problèmes d'alternances et de traitements parallèles

L'utilisation de sémaphores est souvent la manière la plus simple et efficace pour forcer une certaine alternance ou séquence d'exécution entre différentes parties de plusieurs processus.

1.5.1. Alternance stricte: processus Hi et Ho

On veut forcer une alternance stricte entre deux processus: un qui écrit "Hi" et l'autre qui écrit "Ho". On veut donc que le texte imprimé comporte une alternance "Hi Ho Hi Ho ...". Cette alternance doit être respectée peu importe le temps d'exécution de chaque processus (temps qui peut changer avec le temps). Nous définissons donc deux sémaphores, a et b, l'une signalant que "Hi" a fini d'écrire et que c'est maintenant au tour de "Ho" à écrire, l'autre signal l'inverse.

```
                    var  a: semaphore := 1
                    var  b: semaphore := 0
process  Hi                                  process  Ho
begin                                        begin
        P_wait( a )                                  P_wait( b )
        .. ecriture de "Hi"                          .. ecriture de "Ho"
        printf( "Hi ")                               printf( "Ho " )
        V_signal( b )                                V_signal( a )
end Hi                                       end Ho
```

1.5.2. Lecture, traitement et impression en parallèle

Un programme de calcul scientifique est composé de trois parties: Lecture d'un enregistrement de données, Traitement de ces données et Impression des résultats. Ceci est inclu dans une boucle infinie.

On peut accélérer l'exécution en séparant ce programme en trois processus parallèles. On peut ainsi récupérer les temps perdus en entrée et sortie (lors des phases Lecture et Impression).

```
loop
    Lecture
    Traitement
    Impression
end loop
```

```
process P1          process P2          process P3
   loop                loop                loop
      Lecture             Traitement          Impression
   end loop            end loop            end loop
end P1              end P2              end P3
```

Naturellement, ces processus doivent être synchronisés, car on ne doit exécuter Traitement que si les données sont prêtes (à la fin de Lecture), et Impression ne doit être exécuter que si les résultats sont prêts (à la fin de Traitement).

On vous demande de synchroniser ces processus pour les deux cas suivants en utilisant des sémaphores. Pour les sémaphores, on possède un type sémaphores que l'on peut initialisé (par exemple: var exm: sémaphores := 0) et les deux primitives P() et V().

On utilise une zone mémoire partagée pour transmettre les données de P1 à P2, et de P2 à P3. C'est-à-dire que P1 range à la suite dans une zone mémoire ces données et que P2 les retire au fur et à mesure de ces besoins. Le rangement et le prélèvement des informations dans ces zones font partie de Lecture, Traitement et Impression. On ne s'occupera donc pas de leur écriture. On s'occupera seulement de synchroniser les processus.

1er cas: On dispose de zones de mémoire infinie. Ainsi la Lecture peut être exécutée de façon répétitive sans être arrêtée, et le Traitement n'a pas besoin d'attendre après l'Impression.

2e cas: Les zones mémoires pour le transfert de P1 vers P2 et de P2 vers P3 ne peuvent contenir que deux informations à la fois. Par exemple, la lecture de information #3 ne peut commencer que si le Traitement de l'information #1 est terminée. De même, le Traitement de l'information #3 (produisant le résultat #3) ne peut commencer que si l'Impression du résultat #1 est terminé.

SOLUTION

Les sémaphores a et aa pour la synchronisation entre P1 et P2, et b et bb entre P2 et P3.

1er cas)

```
                        var a : semaphores := 0
                        var b : semaphores := 0

process P1              process P2              process P3
   loop                    loop                    loop
                              P_wait(a)                P_wait(b)
      Lecture                 Traitement               Impression
      V_signal(a)             V_signal(b)
   end loop                end loop                end loop
end P1                  end P2                  end P3
```

2e cas)

```
                        var a : semaphores := 0
                        var b : semaphores := 0
                        var aa : semaphores := 2
                        var bb : semaphores := 2

process P1              process P2              process P3
   loop                    loop                    loop
      P_wait(aa)              P_wait(bb)
                              P_wait(a)                P_wait(b)
      Lecture                 Traitement               Impression
      V_signal(a)             V_signal(b)
                              V_signal(aa)             V_signal(bb)
   end loop                end loop                end loop
end P1                  end P2                  end P3
```

1.5.3. La Confiturerie: Synchronisation de processus

Une confiturerie dispose d'une chaîne pour le remplissage des bocaux. Un processus **Bocal** contrôle une machine qui assure l'alimentation (l'arrivé) des bocaux. Un autre processus **Valve** contrôle l'ouverture et la fermeture de la valve d'une unité de remplissage. La valve ne doit être ouverte qu'après l'arrivé d'un bocal. Le processus **Valve** doit utiliser un processus **Horloge** pour mesurer un délai de 30 secondes, après lequel il ferme la valve. Il doit alors signaler au processus **Bocal** que le remplissage est terminé, et qu'il peut enlever le bocal et en placer un nouveau.

```
process Bocal
    loop
        place un
          bocal
        <<synchro>>
        <<synchro>>
        enlève le
          bocal
    end loop
end Bocal
```

```
process Valve
    loop
        <<synchro>>
        ouvre la valve
        <<synchro>>
        <<synchro>>
        ferme la valve
        <<synchro>>
    end loop
end Valve
```

```
process Horloge
    loop
        <<synchro>>

        délai de
          30 sec.

        <<synchro>>
    end loop
end Horloge
```

A) On vous demande de compléter chacun des processus (en remplaçant les lignes indiquées par <<synchro>>) pour assurer l'exécution dans le bon ordre des différentes parties des 3 processus.

Vous devez utiliser des sémaphores. Pour les sémaphores, on possède un type sémaphore que l'on peut initialisé (par exemple: var exm: sémaphore := 0) et les deux primitives **P_wait()** et **V_signal()**.

B) On ajoute une autre machine d'alimentation en bocaux, processus **Bocal_2**. Il faut s'assurer qu'un seul processus à la fois, **Bocal** ou **Bocal_2**, place un bocal dans l'unité de remplissage. De plus, les nouveaux bocaux (processus Bocal_2) sont deux fois plus grands que les anciens. Le processus **Valve** doit donc obtenir un délai deux fois plus long, e.g. 60 sec (pour les nouveaux bocaux seulement). Pour cela, il devra se synchroniser pour fermer la valve après deux itérations du processus **Horloge** (le délai dans Horloge est fixe et ne peut être changé).

On vous demande de modifier les processus pour assurer une bonne synchronisation pour ce nouveau système.

SOLUTION:

A) Les sémaphores peuvent facilement être utilisées pour assurer l'exécution dans le bon ordre des différentes parties des processus.

```
var b_pret  : semaphore := 0
var b_plein : semaphore := 0
var d_debut : semaphore := 0
var d_fin   : semaphore := 0
```

```
process Bocal
    loop
        place un bocal
        V_signal( b_pret)
        P_wait( b_plein)
        enlève le bocal
    end loop
end Bocal
```

```
process Valve
    loop
        P_wait( b_pret)
        ouvre la valve
        V_signal( d_debut)
        P_wait( d_fin)
        ferme la valve
        V_signal( b_plein)
    end loop
end Valve
```

```
process Horloge
    loop
        P_wait( d_debut)

        délai de 30 sec.

        V_signal( d_fin)
    end loop
end Horloge
```

B) On ajoute une sémaphore, exm, et une variable, boc. Le processus Horloge demeure inchangé

```
var b_pret  : semaphore := 0
var b_plein : semaphore := 0
var d_debut : semaphore := 0
var d_fin   : semaphore := 0
var exm     : semaphore := 1
var boc     : UnsignedInt
```

```
process Bocal
    loop
        P_wait( exm)
        place un bocal
        boc := 1
        V_signal( b_pret)
        P_wait( b_plein)
        enlève le bocal
        V_signal( exm)
    end loop
end Bocal
```

```
process Bocal_2
    loop
        P_wait( exm)
        place un bocal
        boc := 2
        V_signal( b_pret)
        P_wait( b_plein)
        enlève le bocal
        V_signal( exm)
    end loop
end Bocal_2
```

```
process Valve
    loop
        P_wait( b_pret)
        ouvre la valve
        V_signal( d_debut)
        if boc =2 then
            V_signal(d_debut)
            P_wait(d_fin)
        end if
        P_wait( d_fin)
        ferme la valve
        V_signal( b_plein)
    end loop
end Valve
```

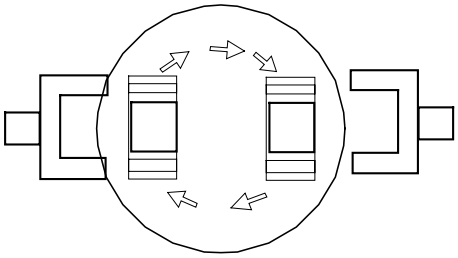
1.5.4. Atelier automatisé - Synchronisation par sémaphores

On vous demande de synchroniser à l'aide de sémaphores trois processus concurrents. Vous disposez d'un type sémaphore que vous pouvez initialiser (par exemple: var exm: sémaphore := 0) et des deux primitives **P_wait()** et **V_signal()**. Vous pouvez également utiliser des variables globales. Il ne faut pas oublier de déclarer les sémaphores et les variables.

Dans un atelier automatisé, un plateau tournant sert au transport des pièces entre deux machines situées de chaque coté du plateau. Le plateau dispose de deux supports pour recevoir les pièces (disposés à 180 degrés), et ne peut tourner que dans une seule direction. Un processus **Plateau** permet de faire tourner le plateau de un demi tour à la fois.
La machine de gauche produit des pièces et les place sur le plateau. Un processus **mGauche** représente cette machine. La machine peut placer la pièce seulement lorsque le plateau est arrêté et que le support de gauche est vide.

La machine de droite retire les pièces du plateau et les consomme. Elle est représentée par un processus **mDroite**. Elle peut retirer une pièce seulement lorsque le plateau est immobile et qu'il y a une pièce sur le support de droite.

Il faut donc faire tourner le plateau après que la machine de gauche y ait placé une pièce et, selon le cas, que la machine de droite ait terminé le retrait d'une pièce. Il faut considérer qu'au début il n'y a aucune pièce sur le plateau.



```
process mGauche
    Loop
        produit une pièce
        <<synchro>>
        place la pièce sur le plateau
        <<synchro>>
    end loop
end mGauche
```

```
process Plateau
    Loop
        <<synchro>>
        tourne le plateau d'un demi tour
        <<synchro>>
    end loop
end Plateau
```

```
process mDroite
    Loop
        <<synchro>>
        retire une pièce du plateau
        <<synchro>>
        consomme la pièce
    end loop
end mDroite
```

A) On vous demande de compléter chacun des processus (en remplaçant les lignes indiquées par <<synchro>>) pour assurer l'exécution dans le bon ordre des différentes parties des 3 processus.

B) Vous devez maintenant considérer le cas où il y a un seul support de pièce sur le plateau. Après avoir placer une pièce sur le plateau, il faut donc 1) tourner le plateau d'un demi tour, 2) retirer la pièce, et 3) tourner le plateau d'un demi tour, avant de pouvoir placer une nouvelle pièce.
Complétez les trois processus pour assurer la synchronisation appropriée.

SOLUTION:

A) Les sémaphores peuvent facilement être utilisées pour assurer l'exécution dans le bon ordre des différentes parties des processus. Les sémaphores d_ma et g_ma indiquent que les machines de droite ou gauche peuvent utiliser le plateau, i.e. placer ou retirer une pièce. Le plateau tourne seulement quand les deux machines ont complétées leurs activités, ce qui est indiqué par les sémaphores d_pl et g_pl.

```
var g_ma  : semaphore := 1
var g_pl  : semaphore := 0
var d_ma  : semaphore := 0
var d_pl  : semaphore := 1
```

```
process mGauche
    Loop
        produit une
          pièce
        P_wait( g_ma )
        place la
          pièce sur
          le plateau
        V_signal( g_pl )
    end loop
end mGauche
```

```
process Plateau
    Loop
        P_wait( g_pl )
        P_wait( d_pl )
        tourne le
          plateau
          d'un demi
          tour
        V_signal( g_ma )
        V_signal( d_ma )
    end loop
end Plateau
```

```
process mDroite
    Loop
        P_wait( d_ma )
        retire une
          pièce du
          plateau
        V_signal( d_pl )
        consomme la
          pièce
    end loop
end mDroite
```

B) On ajoute une variable, tour, pour indiquer quelle machine peut utiliser le plateau, tour = 0 pour la machine de gauche et 1 pour celle de droite. On a besoin de seulement une sémaphore, pl, pour faire tourner le plateau.

```
var g_ma  : semaphore := 1
var d_ma  : semaphore := 0
var pl    : semaphore := 0
var tour  : SignedInt := 0        { 0 - machine de gauche
                                    1 - machine de droite }
```

```
process mGauche
    Loop
        produit une
          pièce
        P_wait( g_ma )
        place la
          pièce sur
          le plateau
        tour := 1
        V_signal( pl )
    end loop
end mGauche
```

```
process Plateau
    Loop
        P_wait( pl )
        tourne le
          plateau
          d'un demi
          tour
        if tour = 0
          then V_signal(g_ma)
          else V_signal(d_ma)
          end if
    end loop
end Plateau
```

```
process mDroite
    Loop
        P_wait( d_ma )
        retire une
          pièce du
          plateau
        tour := 0
        V_signal( pl )
        consomme la
          pièce
    end loop
end mDroite
```

1.5.5. Problème du barbier

The Sleepy Barber Problem [Dijkstra 1965].

A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, then the customers sits in one of the available free chairs. If the barber is a sleep, the customer wakes the barber up. Write a program to coordinate the barber and the customers.

Answer The shared data structures are:

```
var   barber, wait: semaphore {initially = 0}
      entry:  semaphore {initially = 1}
      count:  integer {initially = 0}
```

The code for the barber is:

```
repeat
    P(barber);
    "shave"
until false
```

A customer process executes the following:

```
P(entry);
if count = n then exit;
count := count + 1;
if count > 1 then
    begin
        V(entry);
        P(wait);
    end
else V(entry);
V(barber);
"shave"
P(entry);
count := count - 1;
if count > 0 then V(wait);
V(entry);
```

Problème du barbier avec moniteur
(Turing Plus)

```
monitor Mon
    export suivant, termine, entree_client
    var nb_client : nat := 0   // nombre de client
    var barbier : condition    // pour faire dormir le
                               barbier
    var client  : condition    // pour l'attente des
                               clients

    procedure suivant
        if nb_client = 0 then  wait barbier
            else  signal client  end if
    end suivant

    procedure termine
        nb_client := nb_client - 1
    end termine

    procedure entree_client( var plein: Boolean )
        if nb_client = n  then  plein := true
        else
            plein := false
            nb_client := nb_client + 1
            if empty( barbier) then
                wait  client       // le barbier est
                                   occupe
            else
                signal  barbier    // réveilles le
                                   barbier
            end if
        end if
    end entree_client
end Mon

process processus_barbier
    loop
        Mon.suivant
        // le barbier rase un client
        Mon.termine
    end loop
end processus_barbier

process processus_client    // il y a plusieurs clients
    var plein : Boolean
    Mon.entree_client( plein )
    if plein then exit end if
    // le client se fait raser
end processus_client

fork processus_barbier
loop
    // ...  produit un delai entre les clients
    fork processus_client
end loop
```

1.5.6. Problème des fumeurs

The Cigarette Smokers Problem [Patil 1971].

Consider a system with three smoker processes and one agent process. Each smoker continuously makes a cigarette and smokes it. But to make a cigarette, three ingredients are needed: tobacco, paper, and matches. One of the processes has paper, another tobacco and the third has matches. The agent has an infinite supply of all three. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient can then make and smoke a cigarette, signaling the agent upon completion. The agent then puts out another two of the three ingredients and the cycle repeats. Write a program to synchronize the agent and the smokers.

Answer The shared data structures are

```
var  a:  array [0...2] of semaphore {initially = 0}
       agent:  semaphore {initially = 1}
```

The agent process code is as follows.

```
repeat
   Set i, j to a value between 0 and 2.
   P(agent);
   V(a[i]);
   V(a[j]);
until false;
```

Each smoker process needs two ingredients represented by integers r and s each with value between 0 and 2.

```
repeat
   P(a[r]);
   P(a[s]);
   "smode"
   V(agent);
until false
```

Cette solution n'est pas très bonne car il peut y avoir inter-blocage.

Problème des fumeurs avec moniteur (Turing Plus)

```
type  fumeur_type : enum( tabac, papier, allumette,
                                    aucun )
monitor  mon
    import fumeur_type
    export le_choix, est_ce_moi, fin
    var choix : fumeur_type := fumeur_type.aucun
    var agent : condition
    var fumeur: array fumeur_type of condition

    procedure le_choix( no: fumeur_type)
        // active le fumeur choisis
        choix := no
        signal fumeur(choix)
        // met l'agent en attente
        wait agent
    end le_choix

    procedure est_ce_moi(no: fumeur_type )
        if choix not= no then wait fumeur(no)  endif
        choix := fumeur_type.aucun
    end est_ce_moi

    procedure fin
        signal agent
    end fin
end mon

process Agent
    var no: fumeur_type
    loop
        // choisis les deux items
        no := // le fumeur a activer
        mon.le_choix( no )
    end loop
end Agent

process Tabac
    loop
        mon.est_ce_moi( fumeur_type.tabac )
        fume  // le fumeur avec du tabac fume
        mon.fin
    end loop
end Tabac

process Papier
    loop
        mon.est_ce_moi( fumeur_type.papier )
        fume  // le fumeur avec du papier fume
        mon.fin
    end loop
end Papier

process Allumette
    loop
        mon.est_ce_moi( fumeur_type.allumette )
        fume   // le fumeur avec des allumettes
        mon.fin
    end loop
end Allumette

fork Agent   fork Tabac  fork Papier  fork Allumette
```

1.6. PRODUCTEURS-CONSOMMATEURS OU TAMPON LIMITÉ

Le problème des Producteurs-consommateurs et celui des zones tampons limités sont similaires. Le problème des Producteurs-Consommateurs est présenté dans Silberschatz, page 136. On présente alors la solution par ajout de code spécial dans les programmes. La solution avec sémaphore est présenté dans Silberschatz page 155 et 156. Nous présentons ensuite le problème de la gestion de tampon circulaire (circular buffer) avec Turing Plus (ou OOT). Plus loin, on considère l'allocation de tampon pour de gros messages.

1.6.1. Par ajout de code:

Les processus producteurs-consommateurs sont courants dans les systèmes d'exploitation. Un processus producteur produit de l'information qui est consommée par un processus consommateur. Par exemple, un programme d'impression produit des caractères qui sont consommés par le gestionnaire (driver) d'imprimante. Un compilateur peut produire un code assembleur qui sera consommé par un compilateur assembleur. Celui-ci, à son tour, produit un code objet qui sera consommé par un éditeur de lien (linker) et/ou le module de chargement et d'exécution de programme.

Pour que les producteurs et consommateurs puissent travailler en concurrence, nous avons besoin d'un ensemble de zones mémoire (buffer) qui seront remplies par les producteurs et vidées par les consommateurs. Un producteur peut utiliser une zone pour y placer ses informations tandis qu'un consommateur retire l'information d'une autre zone. Le producteur et le consommateur doivent se synchroniser, pour éviter que le consommateur essaie de retirer des informations qui n'ont pas encore été produites. Le consommateur doit alors attendre que l'item aie été produit.

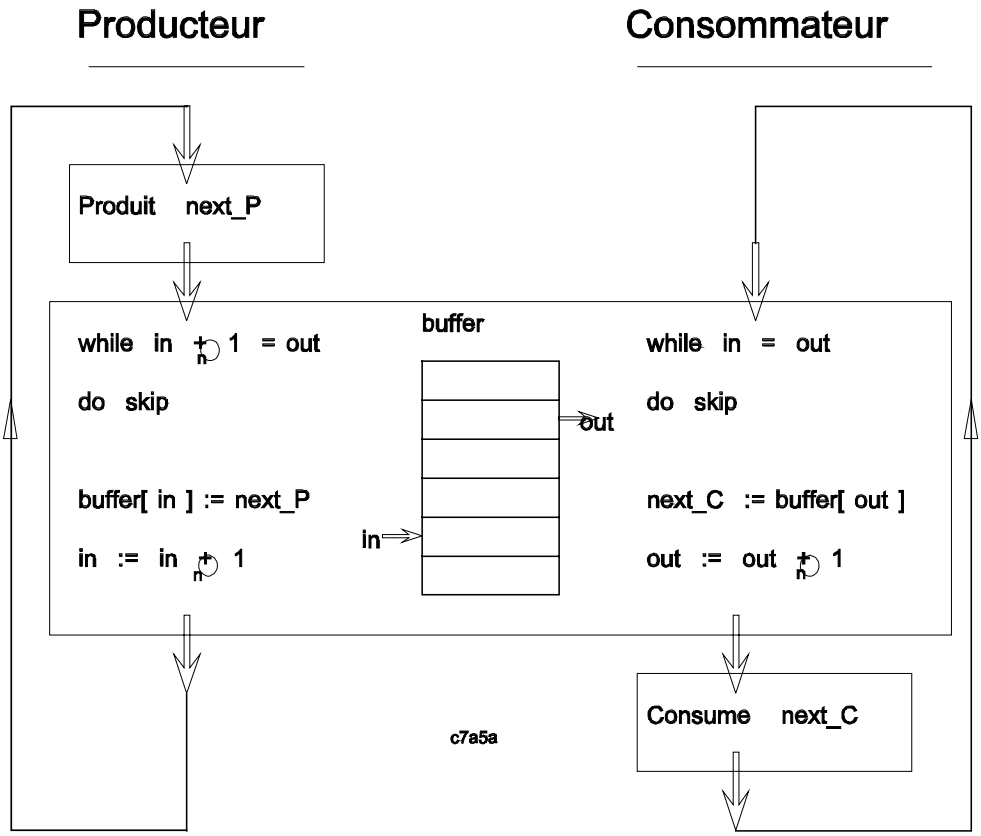
Le cas de zone mémoire illimité (unbounded-buffer) place aucune limite sur le nombre de zone mémoire. Le consommateur peut avoir à attendre pour un nouvel item, mais le producteur peut toujours produire un nouvel item. Il y a toujours une zone libre et disponible. Dans le cas de zone mémoire limité (bonded-buffer), nous disposons seulement d'un nombre fixe et limité de zone mémoire. Comme précédemment, le consommateur doit attendre si les zones sont tous vides. De plus, maintenant le producteur doit aussi attendre si toutes les zones mémoire sont pleines.

Dans la figure qui suit nous présentons la solution pour zone mémoire limité et pour le cas d'un seul producteur et un seul consommateur. Nous disposons de *n* zones mémoire. L'ensemble des zones mémoire est représenté par le vecteur *buffer* qui est partagé entre les deux processus. Cette ensemble est géré comme une liste circulaire, i.e., quand on arrive à la fin du vecteur, on recommence au début. Une opération d'addition sur l'index du vecteur est fait modulo *n*, la taille du vecteur: ($x +_n 1$) correspond à ((x + 1) modulo n). Les variables *in* et *out* sont deux indices identifiant un élément du vecteur *buffer* (une zone mémoire). *in* identifie le prochain élément (zone) libre et *out* le prochain élément plein. Le vecteur est vide quand *in* = *out*; il est plein quand $in +_n 1 = out$. Dans ce cas, toute fois, l'élément *in* n'est pas utilisé et demeure vide tant que l'élément $in +_n 1$ n'a pas été consommé (vidé). L'instruction *skip* est une instruction vide, qui ne fait rien. L'instruction "while *condition* do *skip*" constitue donc une boucle d'attente ou on vérifie continuellement la *condition* jusqu'à ce qu'elle devienne fausse.

La variable *next_P* est utilisée pour emmagasiner l'élément produit avant de le placer dans la zone tampon. De même, *next_C* contient l'élément retiré de la zone tampon et qui est consommé par le processus consommateur.

Remarquons qu'un seul processus modifie la valeur de *in* ou de *out*, et que les deux processus ne travaillent jamais sur la même zone mémoire (élément du vecteur). Il en résulte en particulier

qu'on ne peut utiliser pas plus de *n - 1* zones (élément du vecteur) à la fois. Si toutes les zones étaient utilisées, nous auront *in = out*, ce qui correspond également au cas ou toutes les zones sont vides. Une manière de discerner les deux cas est d'ajouter un compteur indiquant le nombre de zone occupée. Le processus producteur incrémente le compteur lorsqu'il remplit une zone et le processus consommateur le décrémente lorsqu'il vide une zone. Nous avons donc deux processus qui peuvent essayer de modifier la même variable en même temps. Nous aurons alors un problème de section critique (voir section 6.2, premier exemple). Il nous vaudra alors synchroniser les processus pour s'assurer qu'un seul à la fois modifie la valeur du compteur.



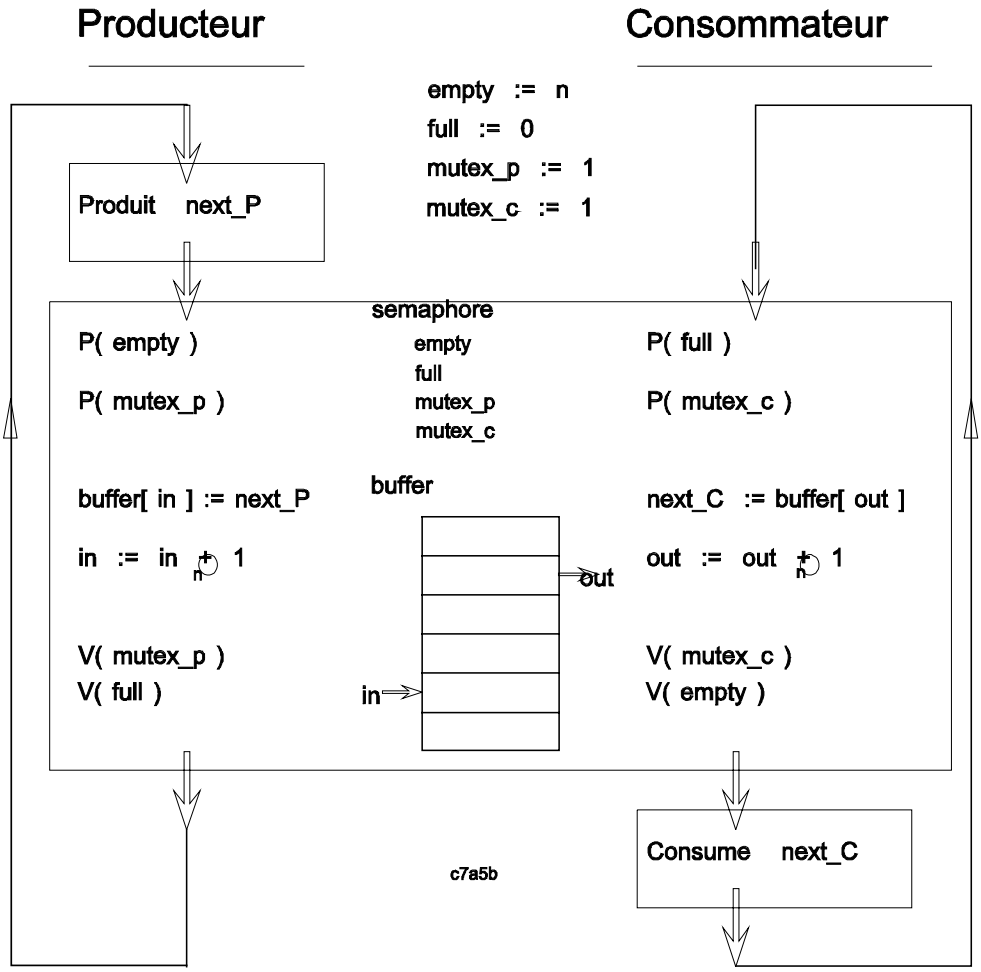
1.6.2. Par sémaphore:

Le problème de section critique associé à l'utilisation d'un compteur peut être résolu en employant un sémaphore à la place du compteur. En fait, nous utilisons deux sémaphores: *full* indiquant le nombre de zones occupées et *empty* indiquant le nombre de zones libres. Les boucles d'attente active, boucles while, sont remplacées par des instructions P-wait. Pour le producteur, on attend qu'il y aie au moins une zone de libre (P(empty)). Pour le consommateur, on attend qu'il y aie au moins une zone d'occuper (P(full)). Le sémaphore *empty* est initialisé à *n*, et le sémaphore *full* est initialisé à 0.

Si nous considérons qu'il peut y avoir plusieurs processus producteurs, nous devons alors nous assurer qu'un seul producteur à la fois utilisera une zone et incrémentera la variable *in*. La copie dans le vecteur *buffer* et l'incrémentation de *in* constituent donc une section critique entre producteur et doit être protégé par un sémaphore d'exclusion mutuelle (*mutex_p*). Ceci s'applique également aux consommateurs. Les sémaphores *mutex_p* et *mutex_c* sont initialisés à la valeur 1.

Remarquez la symétrie entre le programme du producteur et du consommateur. On peut considérer que le producteur produit des zones pleines pour le

consommateur, tandis que le consommateur produit des zones vides pour le producteur.



1.6.3. Avec moniteur: (par message en Turing Plus)

Nous présentons maintenant une solution au problème des producteurs-consommateurs par l'utilisation de moniteur. La solution consiste en fait en l'implantation de la transmission de message par l'utilisation de boîte aux lettres. La boîte aux lettres constitue la zone tampon où le producteur dépose l'information et où le consommateur retire l'information. Les processus producteurs et consommateurs communiquent au moyen de zones tampons partagées. Les producteurs envoient de l'information (messages) aux consommateurs. Les zones tampons sont utilisées comme une queue qui emmagasine ces messages, de telle sorte que n'importe quel processus peut à l'occasion ralentir sans affecter la vitesse des autres. La queue est gérée de manière FIFO (premier entré, premier sorti). Les messages sont donc reçus dans le même ordre qu'ils sont envoyés.

La mémoire tampon est séparée en plusieurs zones (slots), chacune pouvant recevoir un message. Les zones sont réutilisées de manière cyclique. Nous pouvons donc parler de "tampon circulaire" (circular buffer). Le producteur remplie les zones, s'interrompant lorsqu'il n'y a plus de zone libre. Le consommateur vide les zones, s'interrompant lorsqu'il n'y a plus de zone pleine. Le processus *Producer* (voir lignes 50..56) consiste à produire une information et à envoyer cette information par un *send*. Le processus *Consumer* (lignes 60..66) consiste à recevoir l'information par un *Receive* et à traiter cette information.

Nous définissons un moniteur *MailBox* pour l'envoie et la réception de message. Le moniteur possède deux procédures: *Send* et *Receive*. Il gère une seule boîte aux lettres, i.e., une seule queue de message. Il possède un nombre fixe, n, de zones mémoires (ligne 5). L'état de la boîte est défini par trois variables *counter, in* et *out* (lignes 7..9).

Nous utilisons deux conditions *emptySlot* et *fullSlot* (lignes 11..14) correspondant aux deux situations où il y a attente dans le moniteur: 1) quand toutes les zones sont pleines, le producteur doit attendre qu'une zone se libère, 2) quand toutes les zones sont vides, le consommateur doit attendre qu'une zone se remplisse.

La procédure *Send* (lignes 18..32) vérifie d'abord s'il y a une zone libre; si non, il en attend une. Après, on ajoute le nouveau message à la fin de la queue. Finalement, on réveille (signal) un processus suspendu sur la condition *fullSlot* (s'il y en a un), correspondant à un consommateur en attente d'un message.

La procédure *Receive* (lignes 33..47) est similaire. On attend éventuellement sur une condition qu'une zone soit pleine. On vide la zone et signale les producteurs en attente sur la condition *emptySlot*.

Notons que nous avons une seule boîte aux lettres avec un nombre quelconque de producteurs et de consommateurs. Notons également que les opérations de copie entre le message et la zone tampon sont effectuées dans le moniteur, i.e., dans les procédures *Send* et *Receive*.

Si nous voulons que le moniteur puisse gérer plusieurs boîtes aux lettres, il faudra avoir des variables de contrôle (lignes 6 à 14) distingues pour chaque boîte aux lettres. Nous pouvons donc utiliser des vecteurs, ainsi qu'un numéro de boîte indiquant quels éléments des vecteurs correspondent à la boîte. Les procédures *Send* et *Receive* auront un argument de plus identifiant le numéro de la boîte.

```
1   // Message type definition
2   type MessageType :  char ( 100 )

3   monitor MailBox
4       export  Send, Receive

5       const n       :=  20    // number of slots
6       var  slot :    array 0 .. n  of  MessageType
7       var  counter:  0 .. n        // how many full
8       var  in :      0 .. n - 1    // slot for send
9       var  out :     0 .. n - 1    // slot for receive

11      // signaled when  counter  <  n
12      var  emptySlot  :  condition
13      // signaled when  counter  >  0
14      var  fullSlot  :  condition

15      counter  :=  0
16      in   :=  0
17      out  :=  0

18  // SEND :  append a 'message' to the end
                                of mailbox
19  procedure Send ( message : MesssageType )

20      // wait until there is a free slot
21      if  counter  =  n   then
22          wait  emptySlot
23          end if
24      assert  counter  <  n

25      // deposit the message
26      slot ( in )  :=  message
27      in  :=  ( in + 1 ) mod n
28      counter  +=  1

29      // signal receiver that message is pending
30      assert  counter  >  0
31      signal  fullSlot
32  end  Send

33  // RECEIVE :  remove a 'message' from
                                the mailbox
34  procedure Receive ( message :
                                MesssageType )

35      // wait until there is a message available
36      if  counter  =  0   then
37          wait  fullSlot
38          end if
39      assert  counter  >  0

40      // retrive the message
41      message  :=  slot ( out )
42      out  :=  ( out + 1 ) mod n
43      counter  -=  1
```

```
44        // signal sender that there is a free slot
45        assert  counter  <  n
46        signal  emptySlot
47     end  Receive
48  end  MailBox

50  process  Producer
51        var  message  :  MessageType
52        loop
53             produce a message  (an information)
54             MailBox.Send ( message )
55        end loop
56  end  Producer

60  process  Consumer
61        var  message  :  MessageType
62        loop
63             MailBox.Receive ( message )
64             process the message  ...
65        end loop
66  end  Consumer

70  fork  Producer   //  fork  one or many  Producer
71  fork  Consumer //  fork  one or many
                                          Consumer
```

1.6.4. Gestion de grandes zones tampons (messages)

La solution de la section précédente est acceptable si la taille des messages, à copier dans les zones tampons de la boîte aux lettres, est petite. Si la taille des messages ou de l'information à transmettre est très grande, un processus passera trop de temps dans le moniteur à copier l'information. La copie de l'information devrait être fait en dehors du moniteur. Elle pourra alors se faire en parallèle dans plusieurs processus et évitera des attentes inutiles après le moniteur. Dans la solution qui suit, une boîte aux lettres est utilisée par les producteurs pour transmettre aux consommateurs seulement les numéros des zones tampons utilisées. La copie d'un numéro est rapide et ne retarde pas beaucoup le temps d'exécution des procédures du moniteur. De plus, nous utilisons le moniteur *BufferMng* de la section 7.9.2 pour gérer l'allocation des zones tampons. La procédure *acquire* retourne le numéro d'une zone libre, tandis que la procédure *release* est utilisé pour indiquer d'une zone est à nouveau libre.

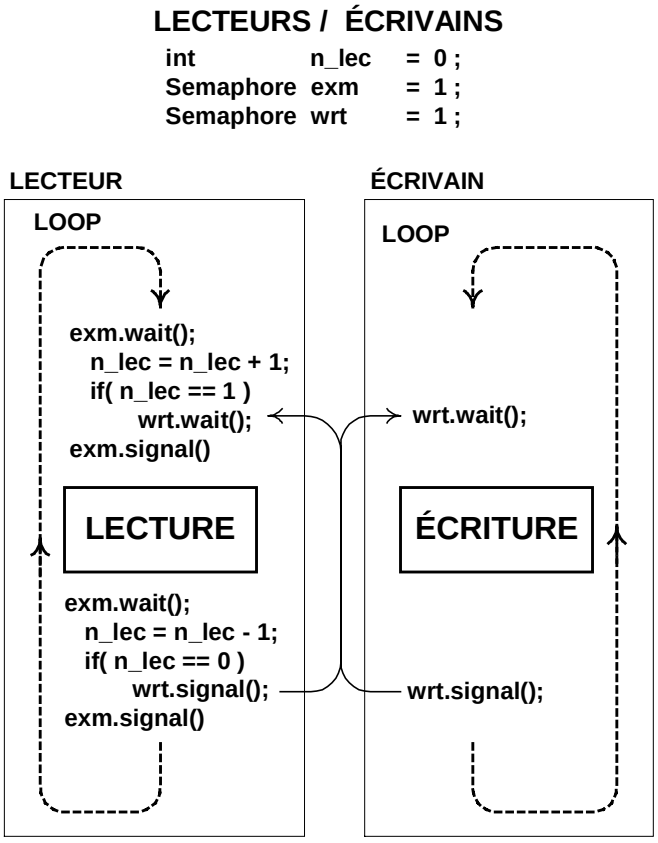
Un processus *Producer* produit d'abord une information. Puis, il obtient un numéro de zone tampon libre. Il copie l'information dans la zone tampon et transmet le numéro de la zone aux consommateurs. Un processus *Consumer* obtient d'abord un numéro de zone tampon des producteurs par un *Receive*. Puis, il copie l'information de la zone dans 'info' et libère la zone. Il traite finalement l'information reçue.

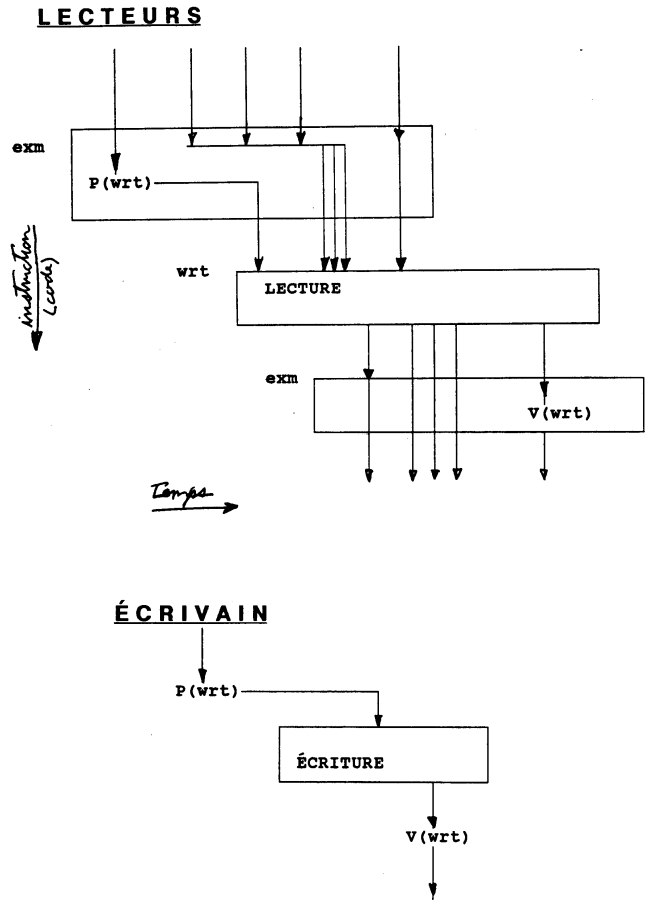
```
// M is the number of shared buffers
const  M     :=  50
include       "BufferMng.t" // voir section 7.9.2

// MailBox  store 'range 0..M-1' value into 'n' slots
type  MessageType:  0 .. M-1     // or int
include "MailBox.t"        // MailBox monitor of 7.5.3

// shared buffers for large messages
type  Information:    char( 500 )  // could be any
                                      type
var   buffer:         array  0 .. M-1  of  Information

process  Producer
     var  buf_no:0 .. M-1
     var  info:     Information
     loop
          ...  exit  if there is no more information  ...
          ...  produce an information into 'info' ...
          BufferMng.acquire( buf_no )
          buffer( buf_no )  :=  info
          MailBox.Send ( buf_no )
     end loop
end  Producer

process  Consumer
     var  buf_no:0 .. M-1
     var  info:     Information
     loop
          MailBox.Receive ( buf_no )
          info  :=  buffer( buf_no )
          BufferMng.release( buf_no )
          ...  process the information in 'info' ...
     end loop
end  Consumer

fork  Producer     //  fork  one or many  Producer
fork  Consumer   //  fork  one or many  Consumer
```

1.7. LES LECTEURS ET LES ÉCRIVAINS

Le problème des lecteurs et des écrivains est présenté dans le livre de Stallings à la page 237 et le livre de Silberschatz, 4e ed., à la page 181.

1.7.1. Solution avec sémaphores





1.7.2. Le problème des lecteurs et des écrivains en Turing Plus

We now consider the problem of several processes concurrently reading and writing the same file. Any number of reader processes (processes accessing but not altering information) may access the file simultaneously. However, any writer process updating the file must have exclusive access to the file; otherwise, inconsistent data may result.

This problem arises in an airline reservations system, for example. Several ground personnel, each using a computer terminal, are issuing boarding passes for a flight. Reservations for the flight have been stored in a file. Reading the file allows an attendant to verify a passenger's reservation. Writing the file allows an attendant to add a new passenger to the flight when a customer arrives at the counter without a prior booking.

The problem, as stated so far, is an extension of the mutual exclusion problem previously discussed. There must be mutually exclusive access to the reservations file. The new feature is that any number of reader processes may be accessing the file simultaneously. However, writers still need exclusive access.

A simple solution to this problem using monitors might be developed as follows. A reader wishing to access the file calls a monitor entry named StartRead. If there is no active writer (there may be several active readers), the number of active readers is increased by 1 and the new reader accesses the file. If there is an active writer, the reader waits. When a reader is awakened from a wait in StartRead, it signals other waiting readers that they also may access the file. A reader finishing accessing the file calls a monitor entry EndRead. It decreases the number of active readers by 1, and if this number reaches zero, it signals a writer that the file is available. (Readers cannot be waiting.) A writer wishing to access the file calls a monitor entry StartWrite. If there is an active writer or at least one active reader, the writer waits. A writer that is finished with the file calls EndWrite, and signals waiting readers or writers that the file is now available.

This simple solution unfortunately has an important defect. Once one reader begins accessing the file, the writers may be indefinitely postponed by a persistent stream of reader processes accessing the file. Some additional restriction on the problem must be made to remove the possibility of indefinite postponement.

We will impose the following requirements on the order of accessing the file. (They make the example more realistic and indefinite postponement is avoided.) We require that a new reader not be permitted to start if there is a writer waiting for the currently active readers to finish. Similarly, we require that all readers waiting at the end of a writer execution be given priority over the next writer. This latter restriction avoids the danger of the indefinite postponement of readers because of many active writers.

We now discuss a solution to the readers and writers problem with these restrictions. We will use a monitor with four entries (the names of these entries are the same as those above, but their structure is different): a reader process calls StartRead before reading and EndRead after reading; a writer process calls StartWrite before writing and EndWrite after writing. The following rules for the entries satisfy our ordering requirements.

StartRead: if there is an active writer or a waiting writer, the reader waits. When a waiting reader is awakened, it signals other readers to become active.

EndRead: If the finishing reader finds that it is the last active reader, it signals a waiting writer.

StartWrite: If there are active readers or if there is an active writer, the new writer-waits.

EndWrite: If there are readers waiting, the finishing writer signals a reader. Otherwise, it signals another writer.

We will now consider these rules for these entries in more detail. In StartRead, a reader waits if either of two situations exists: there is an active writer or

there is a waiting writer. Because a reader in StartRead needs to distinguish between readers and writers in waiting to access the file, there should be separate conditions on which the readers (okToRead) and writers (okToWrite) wait. The empty pre-defined function can be used to test whether there are processes waiting on these conditions. Once a waiting reader in StartRead is resumed, it increases the number of active readers (numberReading) by one. This reader knows that the file is now available for reading, so it signals another reader that was waiting for access to the file. A resumed reader in StartRead thus contributes to a cascade of signaling readers which are waiting. No other process (such as an arriving reader or writer) can enter the monitor while this cascade is in progress. In time, all readers that were waiting for access after a writer are signaled. This version of StartRead differs from our previous version because now a reader waits if there is a waiting writer.

In EndRead, a reader decreases the number of active readers by one. If it finds that it is the last reader accessing the file, it signals a waiting writer. The logic of this entry is therefore identical to that of the previous version of EndRead.

In StartWrite, a writer waits if either of two situations exists: there is at least one active reader or there is an active writer. The variable numberWriting records whether writing is taking place. The logic of this entry is therefore identical to that of the previous version of StartWrite.

In EndWrite, a writer first checks if there are readers waiting to access the file. If there are, the writer signals a waiting reader. If there are not, the writer signals a waiting writer. This again illustrates the need for separate condition variables for readers and writers. EndWrite differs from our previous version because here a writer tries to signal a reader before it signals a writer.

```
// Solution to the Readers and Writers problem

monitor FileAccess

    export( StartRead, EndRead, StartWrite,
                        EndWrite )

    var numberReading: nat : 0
    var numberWriting: 0..1 : 0

    // Signaled when numberWriting = 0
    var okToRead: condition

    // Signaled when numberWriting = 0 and
    //                    numberReading = 0
    var okToWrite: condition

    /* START READ
    If there is an active writer or a waiting writer,
     then block. If awakened then attempt to
    awaken another blocked reader.
    */
    procedure StartRead
        if    numberWriting > 0 or
              not empty(okToWrite) then
                  wait okToRead
        end if
        assert numberWriting=0

        numberReading += 1
        signal okToRead
    end StartRead


    /* END READ
    If no more reader. active then attempt to
    awaken a blocked writer.
    */
    procedure EndRead
        numberReading -= 1
        if numberReading = 0 then
              signal okToWrite
        end if
    end EndRead

    /* START WRITE
    If there are active reader. or if there is an
    active writer, block.
    */
    procedure StartWrite
        if    numberWriting > 0 or
              numberReading > 0 then
                  wait okToWrite
        end if
        assert numberWriting = 0 and
                            numberReading = 0
        numberWriting += 1
    end StartWrite
```

```
    /* END WRITE
    If there are readers waiting then unblock
    one of them.
    Otherwise attempt to unblock a writer.
    */
    procedure EndWrite
        numberWriting -= 1
        if not empty(okToRead) then
            signal okToRead
        else
            signal okToWrite
        end if
    end EndWrite
end FileAccess


process Reader( r: int )
    var reading: int
    loop
        FileAccess.StartRead
        randint( reading,0,100)
        pause reading
        FileAccess.EndRead
    end loop
end Reader

process Writer( v: int )
    var writing: int
    loop
        FileAccess.StartWrite
        randint( writing,0,100)
        pause writing
        FileAccess.EndWrite
    end loop
end Writer

for i: 0..3
    fork Writer( i)
    fork Reader( i)
end for
```

The restrictions of the readers and writers problem require that waiting readers are given priority over waiting writers after a writer finishes, and a waiting writer is given priority over waiting readers after all readers finish. It is interesting to note that this form of "precedence" scheduling using signal statements is accomplished without priority conditions. Simple tests on the variables numberReading and numberWriting, and calls to empty(okToRead) and empty(okToWrite) suffice to achieve the desired order of file access.

Finally, we discuss the signal statement in StartRead. A reader executing this signal suspends execution and allows a waiting reader to enter the monitor; this creates the cascade that activates waiting readers. When the last waiting reader is activated, it will signal an empty condition. The semantics of the signal statement in this situation are that the signaling process continues execution, possibly after other processes have entered the monitor. The correctness of our solution is not affected, however. No writer can intervene during any of the suspended reader executions because the writer first checks on numberReading in the StartWrite entry. In these cases, numberReading is greater than 0, thanks to the increment of nmberReading in StartRead before signaling okToRead.

A reader attempting to enter the monitor by a statement during the cascade of resuming readers will be blocked until the monitor becomes free. At that point, if the reader finds that there is a waiting writer, it will wait; if there is not a waiting writer, it will proceed.

This concludes our discussion of the readers and writers problem. We saw that some complex scheduling decisions could be made without the need for priority conditions. A later example (the clock manager) will show that priority conditions are sometimes convenient.

1.7.3. Traversee de la riviere

Des pierres sont disposées dans le cours d'une rivière pour permettre sa traversée par une ou des personnes venant d'une seule direction à la fois. Désignons par A et B les deux rives de la rivière. Assignons à chaque personne sur la rive A un processus P_A, et à chaque personne sur la rive B un processus P_B.

```
process P_A
        TRAVERSEE
        DE A VERS B
end P_A

process P_B
        TRAVERSEE
        DE B VERS A
end P_B
```

Ecrivez un moniteur (en Euclid Concurrent) et complétez des processus pour assurer la coordination de la traversée entre les personnes venant de rive opposée. La solution doit permettre la traversée simultanée de plusieurs personnes venant d'une même rive, et doit également assurer qu'il n'y aura pas d'interblocage (deadlock).

A) Dans un premier temps, vous ne tiendrez pas compte des risques de famine (report à l'infini, 'starvation'). Recherchez plutôt la simplicité dans l'écriture du moniteur.

B) Dans un second temps, suggérez un approche pour éviter les risques de famine, écrivez le moniteur correspondant, et expliquez comment les risques de famine sont évités.

A) Solution avec risque de famine.

```
monitor  Mon
    export ArriveeA, FinA, ArriveeB, FinB
    var nombreA : nat := 0
    var nombreB : nat := 0
    var suspendA: condition
    var suspendB: condition

    procedure ArriveeA
        if nombreB > 0
            then    wait suspendA
                    signal suspendA
            end if
        nombreA := nombreA + 1
    end ArriveeA

    procedure FinA
        nombreA := nombreA - 1
        if nombreA = 0 then signal suspendB endif
    end FinA

    procedure ArriveeB
        if nombreA > 0
            then    wait suspendB
                    signal suspendB
        end if
        nombreB := nombreB + 1
    end ArriveeB

    procedure FinB
        nombreB := nombreB - 1
        if nombreB = 0 then signal suspendA endif
    end FinB
end Mon

process P_A
    Mon.ArriveeA
    EXECUTION DE LA TRAVERSEE DE A VERS B.
    Mon.FinA
end P_A

process P_B
    Mon.ArriveeB
    EXECUTION DE LA TRAVERSEE DE B VERS A
    Mon.FinB
end P_B

loop
    fork P_A ou P_B
    // Il y a plusieurs processus P_A et P_B
    // qui arrivent à des temps indéterminés
end loop
```

B) Sans risque de famine.

Lors de l'arrivée, un processus P_A (P_B) va être suspendu s'il y a des B (A) en train de traverser ou en attente. Ainsi des processus P_A (P_B) peuvent être suspendu même si des A (B) sont en train de traverser. La fin des A va activer tous les B en attente, et la fin des B tous les A en attente. On a une priorité alternée de A à B. Les procedures suivantes du moniteur sont modifiées:

```
procedure ArriveeA
    if nombreB > 0 or not empty( suspendB )
        then    wait suspendA
                signal suspendA
        end if
    nombreA := nombreA + 1
end ArriveeA
```

```
procedure ArriveeB
    if nombreA > 0 or not empty( suspendA )
        then    wait suspendB
                signal suspendB
    end if
    nombreB := nombreB + 1
end ArriveeB
```

Traversée de la rivière avec sémaphore

Soit A et B les deux rives. La solution suivante est inspirée du problème des lecteurs-écrivains de Peterson. Il y a cependant risque de famine.

```
var ex_A, ex_B, tr : semaphore := 1
var n_A, n_B       : int       := 0
```

process P_A

```
P( ex_A )
n_A := n_A + 1
if n_A = 1 then P( tr ) end if
V( ex_A )

TRAVERSEE DE A

P( ex_A )
n_A := n_A - 1
if n_A = 0 then V( tr ) end if
V( ex_A )
```

process P_B

```
P( ex_B )
n_B := n_B + 1
if n_B = 1 then P( tr ) end if
V( ex_B )

TRAVERSEE DE B

P( ex_B )
n_B := n_B - 1
if n_B = 0 then V( tr ) end if
V( ex_B )
```

Une autre solution est:

```
var exm, att_A, att_B : semaphore := 1
var n_A, n_B          : int       := 0
```

process P_A

```
P( att_A )
P( exm )
n_A := n_A + 1
if n_A = 1 and n_B not= 0
        then    V( exm )
                P( att_A)
        else  V( exm )
        end if
V( att_A );

TRAVERSEE DE A

P( exm )
n_A := n_A - 1
if n_A = 0 and n_B not= 0
        then V( att_B )
        end if
V( exm )
```

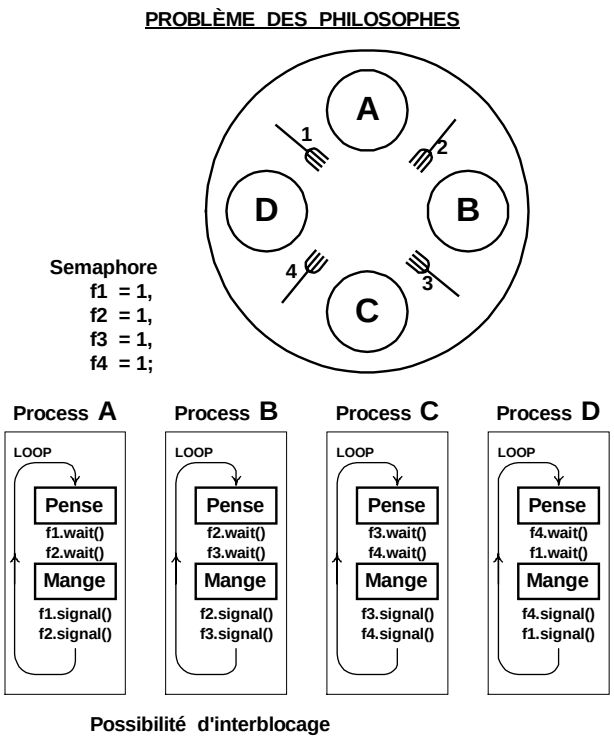
process P_B

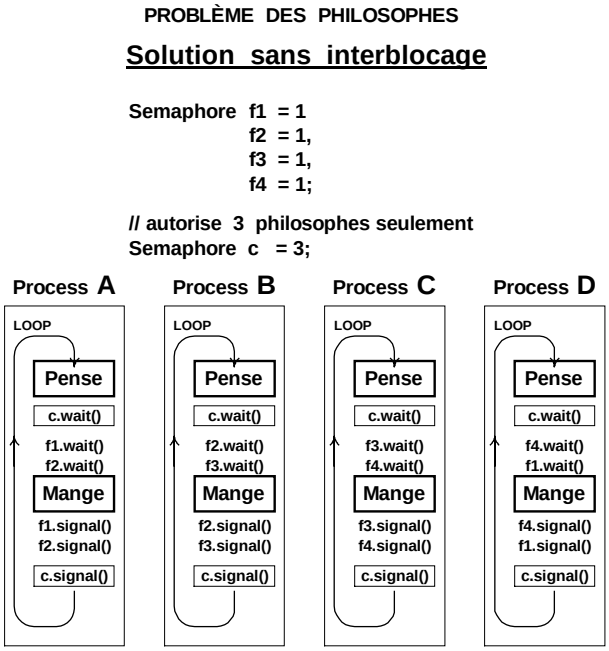
```
P( att_B )
P( exm )
n_B := n_B + 1
if n_B = 1 and n_A not= 0
        then    V( exm )
                P(att_B)
        else V( exm )
        end if
V( att_B )

TRAVERSEE DE B

P( exm )
n_B := n_B - 1
if n_B = 0 and n_A not= 0
        then V( att_A )
        end if
V( exm )
```

1.8. LE PROBLÈME DES PHILOSOPHES

Le problème des philosophes est présenté par Stallings à la page 270 et par Silbertschatz, 4e ed., à la page 183.

1.8.1. Solutions avec sémaphores





1.8.2. Solution avec moniteur en Turing Plus

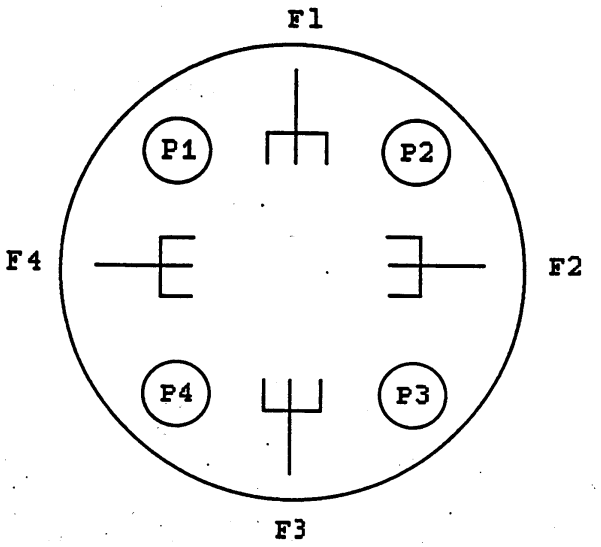
Dining Philosophers

Suppose several processes are continually acquiring, using, and releasing a set of shared resources. We want to be sure that a process cannot be deadlocked (blocked so that it can never be signaled) or indefinitely postponed (continually denied a request).

A colorful version of this problem can be stated in terms of a group of philosophers eating spaghetti. It goes like this:

> There are N philosophers who spend their lives either eating or thinking. Each philosopher has his own place at a circular table, in the center of which is a large bowl of spaghetti. To eat spaghetti requires two forks, but only N forks are provided, one between each pair of philosophers. The only forks a philosopher can pick up are those on his immediate right and left. Each philosopher is identical in structure, alternately eating then thinking. The problem is to simulate the behavior of the philosophers while avoiding deadlock (the request by a philosopher for a fork can never be granted) and indefinite postponement (the request by a philosopher for a fork is continually denied).

We will concentrate on the case of four philosophers. Here is a picture of a table setting with four plates and forks.



Several points should be clear from the problem description. Adjacent philosophers can never be eating at the same time. Also, with four forks and the need for two forks to eat, at most two philosophers can be eating at any one time. Any solution we develop should allow maximum parallelism.

Consider the following proposed solution. A philosopher acquires his or her forks one at a time, left then right, by calling a monitor entry PickUp, giving as a parameter the appropriate fork number. Similarly, a philosopher returns his forks one at a time, left then right, by calling a monitor entry PutDown. The philosopher's activity is represented by a process that repeatedly executes the statements:

```
PickUp( left)
PickUp( right)
Pause eating
PutDown( left)
PutDown( right)
Pause thinking
```

The entries are part of a monitor that controls access to the forks. The data of the monitor includes the one-dimensional boolean array idleForks, where idleForks(i) gives the availability of fork number i Only when a philosopher acquires his two forks does he begin eating. Periods of eating and thinking by a philosopher can be represented by pause statements of appropriate duration.

Unfortunately, this simple solution suffers from a serious defect, namely deadlock. Consider a sequence of process executions in which the philosophers each acquire a left fork, then each attempt to pick up a right fork. Each philosopher will be blocked in the Pick Up entry on a condition that can never be signaled; the request for a right fork can never be granted. Deadlock occurs in this situation because processes hold certain resources while requesting others. Clearly, we need a solution that prevents deadlock.

We now discuss a solution that prevents deadlock. Each philosopher is represented by a process that repeatedly executes the statements

```
PickUp( i)
Pause eating
PutDown ( i)
Pause thinking
```

where i is the number of the philosopher. Picking up the forks is now represented as a single monitor entry call.

We will develop a monitor named Forks with two entries, PickUp and PutDown, which acquire and release tbe forks. The structure of these entries differs from that of the previous entries. The monitor must have variables that keep track of the availability of the four forks. This could be done by having an array of four elements; this is the earlier idleForks method. We will use a different approach. There is still an array of four elements, but the elements will correspond to the four philosophers. The array will be called freeForks, where freeForks(i) is the number of forks available to philosopher i: either 0, 1, or 2. In the PickUp entry, philosopher us allowed to pick up his forks only when freeForks(i)= 2. Otherwise, he waits on the condition ready(i) each philosopher thus has his own condition for which he waits. When he succeeds in picking up his forks, he must decrease the fork counts of his neighbor philosophers. He then leaves the monitor and commences eating.

The neighbors of philosopher i are numbered Left(i) and Right(i). Based on our earlier diagram, Left(2) is 3 and Right(2) is 1. It is important to note that Left (4) is 1 and Right(1) is 4. Left and Right could be implemented as vectors initialized to the appropriate values or as functions using the mod pre-defined function to calculate the proper neighbor number.

When philosophers return their forks in the PutDown entry, they should increase the fork counts of both neighbors. If they then find that a neighbor has two forks available, they should signal the appropriate neighbor. The philosophers therefore pass the ability to access the forks among themselves using signal statements.

```
// Solution to the dining philosopher's problem

type Philosophers: 1..4

monitor Forks
    import( Philosophers )
    export( Pickup, PutDown )

    // Number of forks philosopher i has available.
    var freeForks: array Philosophers of 0..2
    for i: Philosophers
        freeForks(i) := 2
    end for

    // signaled when freeForks(philosopher)=2
    var ready: array Philosophers of condition

    Declaration and initialization of Left and Right

    /* PICK UP
    Attempt to pick up both forks
    */
    procedure PickUp (me: Philosophers)
        // Wait until both forks are available
        if freeForks(me) not= 2 then
            wait ready(me)
        end if
        assert freeForks(me) = 2

        // Tell neighbors that forks are being used
        freeForks( Right(me))   -= 1
        freeForks( Left(me))    -= 1
    end Pickup

    /* PUT DOWN
    Return the forks to the table
    */
    procedure PutDown(me: Philosophers)
        freeForks( Right(me)) += 1
        if freeForks( Right(me)) = 2 then
            signal ready( Right(me))
        end if

        freeForks( Left(me)) += 1
        if freeForks( Left(me)) = 2 then
            signal ready( Left(me))
        endif
    end PutDown
end Forks
```

We now examine the solution for deadlock and indefinite postponement. Deadlock would occur if a philosopher became blocked and could not continue executing regardless of the future of the

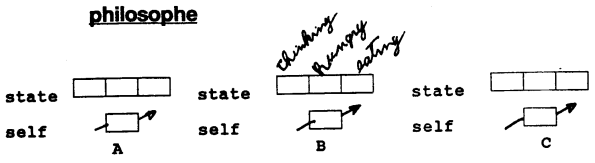
system. Let us look at the program to see where this might occur. A philosopher cannot become blocked forever when eating or thinking. Therefore, we look at execution in the two monitor entries to see if deadlock can occur there. The PickUp entry shows that a requesting philosopher suspends his execution when his two forks are not available. The philosopher does not pick up one fork and wait for the other. This means that the system can never get into the state where each philosopher holds one fork (the left one, say) and is waiting for the other --- this is deadlock, brought about by holding resources while requesting others. The rest of the PickUp entry shows that the fork counts are correctly decreased. The PutDown entry increases the fork counts and correctly signals other waiting philosophers.

The above discussion shows informally that deadlock cannot occur in our solution. Indefinite postponement is a different matter. Indefinite postponement occurs if a philosopher becomes blocked and there exists a future execution sequence in which he will remain forever blocked. Consider the following situation for philosophers 1, 2, and 4 in which philosopher 1 is indefinitely postponed.

Suppose philosopher4 picks up two forks (forks 3 and 4). Next, philosopher 2 picks up two forks (forks 1 and 2). Next, philosopher 1 enters PickUp and finds that his required forks are being used, so he waits. Next, the following unfortunate sequence occurs repeatedly. Philosopher 4 puts down both forks, thinks, and then picks them up again; then, philosopher 2 puts down both forks, thinks, and picks them up again, and so on. During this repeated sequence, at reast one of the forks of philosopher 1 is always being used. Given that it is possible for this sequence to repeat indefinitely, we see that philosopher 1 can suffer indefinite postponement. We have not solved the problem!

There are several ways to overcome this defect. The most obvious one keeps track of the “age” of requests for forks, and when one request gets too old, other requests are held up until the oldest request can be satisfied. This could be done, for example, by counting the number of meals enjoyed by a philosopher’s two neighbors while he or she is waiting for their forks. If a philosopher is bypassed more than, say, 10 times, his neighbors are blocked until that philosopher gets a chance to pick up his forks. (Does this solution allow maximum parallelism?) We leave this extension to the reader.

1.8.3. Une autre solution avec moniteur

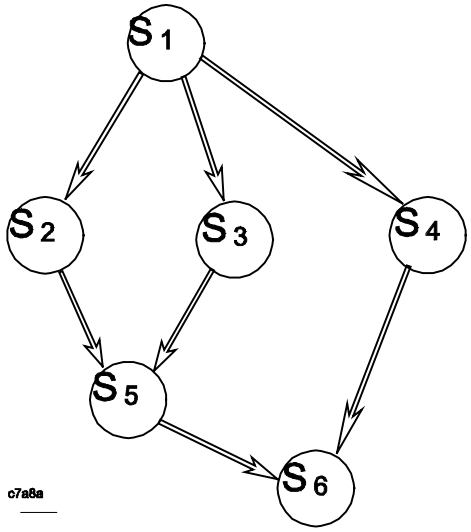



```
process philosophe( i )
        loop
                THINK
                dp.pickup(i);
                EAT
                dp.putdown(i);
        end loop;

monitor dp  // dining-philosophers
  var state: array[0..4] of (thinking, hungry, eating);
  var self:  array[0..4] of condition;

  procedure entry pickup (i:0..4);
    begin
      state[i]:= hungry;
      test (i);
      if state[i] not= eating then self[i].wait;
    end;

  procedure entry putdown (i:0..4);
    begin
      state[i]:= thinking;
      test (i-i mod 5);
      test (i+l mod 5);
    end;

  procedure test (k:O..4);
    begin
      if state[k-1 mod 5] not= eating
        and state[k] = hungry
        and state[k+1 mod 5] not= eating
        then begin
          stage[k]:= eating;
          self[k].signal;
        end;
    end;

  begin
    for i: 0 to 4
      do state[i]:= thinking;
  end.
```

1.9. PROBLEMES AVEC SÉMAPHORES

1.9.1. Graphe de préséance

Programmez le graphe de préséance de la figure suivante. Les instructions *parbegin* et *parend* indiquent l'exécution en parallèle de chacune des instructions (ou groupes d'instructions) qu'ils contiennent.



a) avec parbegin et parend

```
S1;
parbegin
        S4;
        begin
                parbegin
                  S2;
                  S3;
                parend
                S5;
        end
parend
S6;
```

b) avec fork et join

```
        C1 := 2;
        C2 := 2;
        S1;
        fork L1;
        S4;
        goto L4;
L1:     fork L2;
        S2;
        goto L3;
L2:     S3;
L3:     join C1;
        S5;
L4:     join C2;
        S6;
```

1.9.2. Graphe de préséance avec sémaphores

Considérons le graphe de préséance de 1.9.1. Supposons que nous ajoutons le lien (S_2,S_4) au graphe.

a. Est-ce-que ce nouveau graphe de préséance peut être exprimé seulement à l'aide des instructions de concurrence *parbegin* et *parend*? Si oui, dites comment; si non, expliquez pourquoi.

Réponse Non

b. Comment pouvez vous implanter ce graphe de préséance si des sémaphores peuvent aussi être utilisés?

var a, b, c, d, e, f, g, h: Semaphore {initially = 0}

```
parbegin
  begin S1;   V(a);   V(b);   V(c);   end;
  begin P(a); S2;     V(d);   V(e);   end;
  begin P(b); S3;     V(f);           end;
  begin P(c); P(d);   S4;     V(g);   end;
  begin P(e); P(f);   S5;     V(h);   end;
  begin P(g); P(h);   S6;             end;
parend
```

1.9.3. Sémaphores binaires

Un sémaphore binaire est un sémaphore dont la valeur entière peut prendre seulement les valeurs 0 ou 1. Montrez comment un sémaphore générale peut être implanté avec des sémaphores binaires.

Réponse Soit S sémaphore générale. Pour l'implanter à partir de sémaphores binaires nous avons besoin des variables suivantes:

```
var  SI: binary-semaphore {initially = 1}
     S2: binary-semaphore {initially = 0}
     C:  integer  {initially = value of S}

P(S)  P( S1 );
      C := C - 1:
      if C < 0  then begin
                  V( S1 );
                  P( S2 );
            end
      else  V( S1 );

V(S)  P( S1 ):
      C := C + 1;
      if C ≤ 0  then  V( S2 );
      V( S1 );
```

1.9.4. Allocation de ressources par ENQ et DEQ

Deux primitives de synchronisation, ENQ et DEQ, sont définies ci-dessous. *r* désigne une ressource, *p* désigne un processus. *queue(r)* est une queue FIFO de processus attendant pour une ressource *r*, et *inuse(r)* est une variable boolienne.

```
ENQ( r):   if inuse( r ) then begin
                  insert  p in queue( r );
                  block p;
            end
           else inuse( r )  := true;

DEQ( r):   p  := head of queue( r);
           if  p ≠ nil  then  activate p
           else inuse( r )  := false;
```

Construisez une implantation de ENQ/DEQ en utilisant les sémaphores. Assurez-vous que l'ordre implécite dans la réactivation des processus is correctement réalisé. Utilisez n'importe quelles structures de données et variables que vous avez besoin.

Réponse: Nous supposons que le système peut supporter jusqu'à N processus en même tamps, chacun avec un identificateur entier unique comprit entre 0 et n-1.

```
var  X:        array[0..n-1] of integers;
     Y:        array[0..n-1] of semaphores
                              {initially = 0}
     in, out:  integer       {initially = 0}
     inuse:    boolean       {initially = false}
     mutex:    semaphore     {initially = 1}

ENQ( r)  P( mutex);
         if inuse  then begin
                        X[ in] := id;
                        in := in + 1 mod n;
                        V( mutex);
                        P( Y[id]);
                  end
         else begin
                  inuse := true;
                  V( mutex);
              end

DEQ( r)  if in ≠ out then begin
                  t := X[ out];
                  out := out + 1 mod n;
                  V( Y[t]);
              end
         else inuse := false;
         V( mutex);
```

1.9.5. Sémaphore multiple

Un sémaphore multiple permet d'effectuer des primitives **P** et **V** sur plusieurs sémaphores en même temps. Il est utile pour acquérir et relâcher plusieurs ressources en une seule opération atomique. Donc, la primitive **P** (pour deux sémaphores) peut être définie comme suit:

```
P( S,R):    while( S ≤ 1 or R ≤ 1) do skip;
            S := S - 1;
            R := R - 1:
```

Montrez comment un sémaphore multiple peut être implanté en utilisant des sémaphores régulières.

Réponse: Les variables partagées sont :

```
var  S1, R1:  integer;
     mutex:   semaphore;  {initially = 1}
     X:       semaphore;  {initially = 0}
```

```
P( S,R)   P( mutex):
          S1 := S1 - 1;
          R1 := R1 - 1;
          if S1 < 0 or R1 < 0  then
                  begin
                          V( mutex);
                          P( X);
                  end
          else  V( X);
```

```
V( S,R)   P( mutex);
          S1 := S1 + 1;
          R1 := R1 + 1;
          if S1 ≤ 0 and R1 ≤ 0  then  V( X);
          V( mutex);
```

1.10. PROBLEMES AVEC MONITEURS

1.10.1. Allocation d'espace mémoire

Nous voulons définir un module (ou une classe) pour gérer la distribution d'espace mémoire ou d'emmagasinement de taille fixe (tel que les pages de la mémoire centrale ou les blocs sur une unité de disque). Les informations sur les espaces libres sont gardées dans un vecteur binaire, *free*. La gestion de ce vecteur est encapsulée dans le module *frames* qui suit. Un processus désirant obtenir une page libre devra exécuter l'instruction "frames.acquire(in)". *in* contiendra alors le numéro de la page libre. S'il n'y a pas de page libre, alors on reçoit la valeur -1 pour *in*. Le processus devra alors soit se mettre en attente ou demander un transfert de page.

Plusieurs processus peuvent cependant appeler les procédures *acquire* et *release* en même temps. Ceci peut rendre les données inconsistantes, par exemple, deux processus peuvent trouver la même page libre. Il faut donc qu'un seul processus n'exécute une procédure à la fois (exclusion mutuelle). Pour cela, nous pouvons utiliser un moniteur: remplacez le mot "module" par "monitor" dans l'exemple.

```
module  frames      // ou monitor frames
    // M is the number of resources (pages)
    var free: array 1 .. M of boolean

    for k : 1 .. M
        free( k ) := true
    end for

    procedure acquire ( var index: int )
        for k :  1 .. M
            if free( k ) then
                free( k ) := false
                index := k
                return
            end if
        end for
        index := - 1
    end acquire

    procedure release ( index: int )
        free(index)  :=  true
    end release
end frames
```

Au lieu de retourner la valeur -1 lorsqu'il n'y a pas de page libre, nous pouvons modifier la procédure *acquire* pour qu'elle suspende le processus. Nous ajoutons donc une variable condition *freePage* et un compteur *count* indiquant le nombre de page libre.

```
monitor  frames
    // M is the number of resources (pages)
    var free:     array 1 .. M of boolean
    var freePage:   condition    // waiting queue
    var count:   int

    count :=  M
    for k : 1 .. M
        free( k ) := true
    end for

    procedure acquire ( var index: int )
        if  count  <=  0  then
            wait  freePage
        end if
        for k :  1 .. M
            if free( k ) then
                free( k ) := false
                index := k
                count -= 1
                return
            end if
        end for
    end acquire

    procedure release ( index: int )
        free(index)  :=  true
        count  +=  1
        signal  freePage
    end release
end frames
```

1.10.2. Allocation de zone tampon

Nous présentons une autre solution au problème 7.9.1. Au lieu de considérer l'allocation de pages mémoire, nous considérons l'allocation de zones tampons (buffer). Nous avons donc M ressources (zones tampons) identifiées par un numéro, *index*, de 0 à M - 1. Nous présentons donc un moniteur *BufferMng* donc la procédure *acquire* retourne le numéro d'une zone libre. La procédure *release* est utilisée pour indiquer d'une zone est à nouveau libre. La liste des zones libres est contenue dans un

vecteur nommé *stack*, qui est géré comme une pile. *top*, le haut de la pile, indique la prochaine zone tampon à utiliser. Si *top* est égale à 0, alors il n'y a plus de zone disponible, et le processus appelant doit être suspendu sur la condition *freeBuffer*. L'initialisation de la pile indique que toutes les zones sont libres.

```
monitor  BufferMng
    // M is the number of resources (buffers)
    const  M     := 50
    var stack:   array 0 .. M-1 of int
    var freeBuffer:  condition     // waiting queue
    var top: int

    top :=  M
    for k : 0 .. M-1
        stack( k ) :=  k
    end for

// ACQUIRE  -  wait until there is a free buffer and
                        return its index
    procedure ***acquire*** ( var index: int )
        if  top  =  0  then
            wait  freeBuffer
        end if
        assert  top  >  0

        top  -=  1
        index :=  stack( top )
    end acquire

// RELEASE  -  return buffer 'index' to the stack of
                        free buffers
    procedure ***release*** ( index: int )
        stack( top )  :=  index
        top  +=  1

        assert  top  >  0
        signal  freeBuffer
    end release
end BufferMng
```

1.10.3. Accès à un fichier

Un fichier doit être partagé entre plusieurs processus, chacun ayant un numéro d'accès unique. Le fichier peut être utilisé par plusieurs processus simultanément mais tout en respectant la contrainte suivante: la somme des numéros d'accès de tous les processus utilisant le fichier à un moment donné doit être inférieure à un seuil N. Écrivez un moniteur pour coordonner l'accès à ce fichier.

```
monitor  coordinator
    var count:        int
    var lowCount:   condition

    count  :=  0
    procedure  acquire( id: int )
        loop
            exit  when  count + id <  N
            wait  lowCount
        end loop
        count  :=  count  +  id
    end acquire

    procedure  leave ( id:  int )
        count  :=  count - id
        signal  lowCount
    end  leave
end  coordinator
```

Pourquoi l'instruction "wait lowCount" est-elle dans une boucle? Peut-elle être placée en dehors de la boucle?

1.10.4. Allocation d'imprimante avec priorité

Nous considérons un système composé de processus P_1, P_2, ..., P_n, chacun ayant un numéro de priorité unique, id. Écrivez un moniteur qui gère l'allocation de trois imprimantes identiques à ces processus, en utilisant le numéro de priorité pour décider de l'ordre d'allocation (une faible valeur indiquant une plus forte priorité). Remarquez l'utilisation d'une variable condition avec priorité pour gérer la priorité des processus.

```
monitor  printer
    var prt:       array 0 .. 2  of boolean
    var freePrinter:  condition  priority

    prt(0) := prt(1) := prt(2) := false

    procedure  acquire ( id: int,  printer_no: int )
        if prt(0) and prt(1) and prt(2)  then
            wait  freePrinter, id
            end if
        if not prt(0)  then          printer_no  :=  0
          elsif not prt(1) then      printer_no  :=  1
          else                   printer_no  :=  2
          end if
        prt(printer_no)  :=  true
    end  acquire

    procedure  release ( printer_no: int )
        prt(printer_no)  :=  false
        signal  freePrinter
    end  release
end  printer
```

1.10.5. Déplacement entre deux salles

Dans un hôtel, deux grandes salles, A et B, sont séparées par une porte étroite qui peut être franchie que par une seule personne à la fois. Une personne de la salle A qui veut franchir la porte est représentée par un processus **PersonneA**. De même, une personne de la salle B qui veut passer dans la salle A est représentée par un processus **PersonneB**. Il peut y avoir un nombre indéterminé de processus A et B (i.e. de personnes). Il peut y avoir qu'une seule personne qui franchit la porte à la fois.

process **PersonneA**

```
.. arrive devant la
..   porte
<<synchro>>
.. franchit la porte
..
<<synchro>>
..
end PersonneA
```

process **PersonneB**

```
.. arrive devant la
..   porte
<<synchro>>
.. franchit la porte
..
<<synchro>>
..
end PersonneB
```

A) Lorsqu'il y a des personnes dans les deux salles en attente, il faut assurer une alternance stricte entre le passage des personnes de A et B. C'est à dire, qu'après le passage d'une personne de A, il faut faire passer une personne de B s'il y en a en attente; si non on continue avec les personnes de A. Même chose pour B.
Ecrivez un moniteur et complétez les processus PersonneA et PersonneB afin d'assurer la synchronisation de leurs activités.

B) Afin d'accélérer le passage, lorsqu'il y a des personnes en attente dans les deux salles, jusqu'à dix (10) personnes d'une même salle peuvent passer de suite (une après l'autre) avant que ce soit le tour des personnes de l'autre salle de passer.
Modifiez ou écrivez un moniteur pour assurer cette synchronisation.

SOLUTION: (Turing Plus)

A) Les procédures Arrive et Quite controlent l'entrée dans la porte. Deux conditions sont utilisées pour suspendre les processus.

```
monitor  Porte
        export ArriveA, QuiteA, ArriveB, QuiteB

        var occupe : boolean := false
        var fileA  : condition
        var fileB  : condition
```

```
        procedure ArriveA
                if occupe then
                        wait fileA  end if
                occupe := true
        end ArriveA

        procedure QuiteA
                occupe := false
                if not empty( fileB)
                        then signal fileB
                else   signal fileA
                end if
        end QuiteA
end Porte
```

```
        procedure ArriveB
                if occupe then
                        wait fileB  end if
                occupe := true
        end ArriveB

        procedure QuiteB
                occupe := false
                if not empty( fileA)
                        then signal fileA
                else   signal fileB
                end if
        end QuiteB
```

```
        process PersonneA
            import Porte

                .. arrive devant la
                ..   porte

                Porte.ArriveA

                .. franchit la porte
                ..

                Porte.QuiteA
                ..
        end PersonneA

        fork PersonneA      //plusieurs
```

```
        process PersonneB
            import Porte

                .. arrive devant la
                ..   porte

                Porte.ArriveB

                .. franchit la porte
                ..

                Porte.QuiteB
                ..
        end PersonneB

        fork PersonneB      //plusieurs
```

B) Deux variables sont ajoutées pour compter le nombre de personnes de A ou B qui ont traversées de suite la porte.

```
var nbA : SignedInt := 0
var nbB : SignedInt := 0
```

Il faut aussi modifier les procédures QuiteA et QuiteB.

```
        procedure QuiteA
                occupe := false
                if empty( fileB )
                    then nbA := 0 end if
                if not empty( fileA) and
                  nbA < 10  then
                        nbA := nbA + 1
                        signal fileA
                else
                        nbA := 0
                        signal fileB
                end if
        end QuiteA
```

```
        procedure QuiteB
                occupe := false
                if empty( fileA )
                    then nbB := 0 end if
                if not empty( fileB) and
                  nbB < 10  then
                        nbB := nbB + 1
                        signal fileB
                else
                        nbB := 0
                        signal fileA
                end if
        end QuiteB
```

1.10.6. Gestion des ventes et inventaires

Un grand magasin utilise un ordinateur central pour gérer ses ventes et son inventaire. Deux processus, **GestionVente** et **GestionInventaire**, doivent donc recevoir et traiter les transactions provenant de deux types de terminal: de vente (caisse) ou de gestion des inventaires. Chaque terminal est représenté par un processus, **TermVente** ou **TermInventaire**.

Une région de mémoire partagée est utilisée pour le passage des transactions entre les processus terminaux et les processus de gestion. La région mémoire est divisée en 10 zones, chacune pouvant contenir une transaction à la fois. Le problème porte donc sur l'allocation de ces zones mémoire à chacun des processus.

A) Vous devez écrire un moniteur pour gérer l'allocation des zones mémoires et contrôler (synchroniser) l'exécution des différents processus. Complétez également le code des processus (appel au moniteur).

- Les processus **GestionVente** ne peuvent traiter que les transactions de vente produites par **TermVente**. De même, **GestionInventaire** ne peuvent traiter que les transactions d'inventaire produites par **TermInventaire**.
- Les processus de gestion doivent être suspendus s'il n'y a pas de transaction à traiter.
- Les processus terminaux doivent être suspendus s'il n'y a pas de zones disponibles pour écrire la transaction.
- Les processus gestion doivent être avisés s'il y a des transactions à traiter.
- Les processus terminaux doivent savoir s'il y a des zones libérées par les processus de gestion.
- Les processus ne doivent pas attendre inutilement.

```
process TermVente
        (ou TermInventaire)
      var no: nat
 loop
      ..recoit transaction
      .. du terminal
      <<synchro>>
      ..écrit transaction
      .. dans la zone #no..
      <<synchro>>
 end loop
end TermVente
```

```
process GestionVente
      (ou GestionInventaire)
      var no: nat
 loop
      <<synchro>>
      ..lit la transaction
      .. de la zone #no...
      <<synchro>>
      ..traitement de la
      .. transaction
 end loop
end GestionVente
```

Au besoin, les appels au moniteur contiendront un paramètre, no, correspondant au numéro (de 1 à 10) de la zone mémoire dans laquelle on peut écrire ou lire une transaction, ou que l'on veut libérer.

Pour simplifier la programmation, vous pouvez définir une ou plusieurs queues FIFO (premier entré, premier sorti) et ajouter ou enlever un élément à une queue (à ne pas confondre avec les queues des conditions). Pour cela, le type **queue_fifo** et les procédures (fonctions) suivantes sont prédéfinis (vous pouvez les employer librement).

```
type->      var q: queue_fifo              {identificateur d'une queue}
            var element: UnsignedInt       {élément de la queue}
```

procédure-> ajoute_au_fifo(q, element)

procédure-> enleve_du_fifo(q, var element)
{vous devez vous assurer que la queue n'est pas vide
avant d'enlever un élément, ex. utilisez un compteur}

B) En plus des contraintes de A), vous devez maintenant assurer un partage équitable des zones mémoires lorsqu'il y a beaucoup de transactions à traiter, c'est-à-dire, lorsque plusieurs terminaux de vente et d'inventaire sont en attente simultanément après une zone mémoire. Apportez les modifications nécessaires au moniteur de A).

Lorsqu'il y a plusieurs **TermVente** et **TermInventaire** en attente, et qu'un processus de gestion libère une zone mémoire, il faut alors allouer la zones de manière à tendre vers un partage égale (i.e. 5 pour les ventes et 5 pour les inventaires).

Naturellement, s'il y a seulement un type de terminal en attente, ce type obtient les zones libres nécessaires, brisant ainsi le partage équilibré des zones.

L'utilisation de toutes les zones mémoires (saturation) résulte du fait que les transactions arrivent plus vite qu'elles peuvent être traitées par les processus de gestion. Si alors un processus de gestion est plus lent que l'autre, il peut avoir tendance à retenir toutes les zones mémoires. Il faut également considérer le cas où un type de terminal reçoit beaucoup plus de transactions que l'autre type.

SOLUTION:

A) La procédure AllouеX obtient une zone libre, EnvoitX et RecoitX avertit le processus de gestion qu'une transaction X doit être traitée, LibereX relache la zone mémoire.

```
moniteur  Zone
        export AlloueVente, EnvoitVente, RecoitVente, LibereVente,
                AlloueInven, EnvoitInven, RecoitInven, LibereInven

        var q_vente: queue_fifo
        var q_inven: queue_fifo
        var q_libre: queue_fifo
        var nb_vente: nat := 0
        var nb_inven: nat := 0
        var nb_libre: nat := 10
        var t_vente, g_vente: condition
        var t_inven, g_inven: condition

        for  i: 1..10
                ajoute_au_fifo( q_libre, i)
        end for

    procedure AlloueVente( var no: nat )
            if nb_libre = 0 then
                    wait( t_vente) end if
            enleve_du_fifo( q_libre, no )
            nb_libre := nb_libre - 1
    end AlloueVente

    procedure AlloueInven( var no: nat )
            if nb_libre = 0 then
                    wait t_inven end if
            enleve_du_fifo( q_libre, no )
            nb_libre := nb_libre - 1
    end AlloueInven
```

```
procedure EnvoitVente( no: nat)
        nb_vente := nb_vente + 1
        ajoute_au_fifo( q_vente, no)
        signal g_vente
end EnvoitVente

procedure RecoitVente( var no: nat)
        if nb_vente = 0 then
                wait g_vente  end if
        enleve_du_fifo( q_vente, no)
end RecoitVente

procedure LibereVente( no: nat)
        nb_vente := nb_vente - 1
        nb_libre := nb_libre + 1
        ajoute_au_fifo( q_libre, no)
        if empty( t_vente)
          then signal t_inven
          else     signal t_vente end if
end LibereVente
end Zone
```

```
process TermVente
        import Zone
        var no: nat
  loop
        ..recoit transaction
        .. du terminal

        Zone.AlloueVente( no)

        ..écrit transaction
        .. dans la zone #no..

        Zone.EnvoitVente( no)
  end loop
end TermVente

process TermInventaire
        import Zone
        var no: nat
  loop
        ..recoit transaction
        .. du terminal

        Zone.AlloueInven( no)

        ..écrit transaction
        .. dans la zone #no..

        Zone.EnvoitInven( no)
  end loop
end TermInventaire
```

```
procedure EnvoitInven( no: nat)
        nb_inven := nb_inven + 1
        ajoute_au_fifo( q_inven, no)
        signal g_inven
end EnvoitInven

procedure RecoitInven( var no: nat)
        if nb_inven = 0 then
                wait g_inven
        enleve_du_fifo( q_inven, no)
end RecoitInven

procedure LibereInven( no: nat)
        nb_inven := nb_inven - 1
        nb_libre := nb_libre + 1
        ajoute_au_fifo( q_libre, no)
        if empty( t_inven)
          then signal t_vente
          else signal t_inven end if
end LibereInven
```

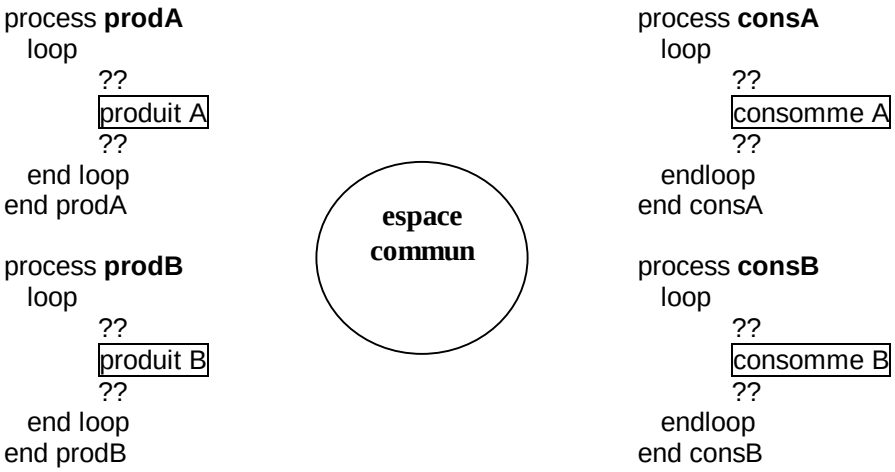
```
process GestionVente
        import Zone
        var no: nat
  loop
        Zone.RecoitVente( no)

        ..lit la transaction
        .. de la zone #no...

        Zone.LibereVente( no)

        ..traitement de la
        .. transaction
  end loop
end GestionVente

process GestionInventaire
        import Zone
        var no: nat
  loop
        Zone.RecoitInven( no)

        ..lit la transaction
        .. de la zone #no...

        Zone.LibereInven( no)

        ..traitement de la
        .. transaction
  end loop
end GestionInventaire
```

B) Avec la solution présentée en A), on assure un partage équitable des zones de mémoires en modifiant une ligne dans LibereVente et LibereInven:

dans LibereVente	on remplace:	if empty(t_vente)
	par:	if empty(t_vente) or nb_vente >= 5
dans LibereInven	on remplace:	if empty(t_inven)
	par:	if empty(t_inven) or nb_inven >= 5

1.10.7. Espace de travail partagé entre machines

Dans une usine, nous avons quatre machines qui partagent un même espace de travail. Deux machines produisent des pièces, le **prodA** produit les pièces **A** et le **prodB** produit les pièces **B**. Les deux autres machines consomment les pièces produites, le **consA** pour les pièces A et le **consB** pour les pièces B.



Il ne peut y avoir que des pièces d'un seul type (A ou B) dans l'espace commun à la fois. Un seul des producteurs peut y placer des pièces. Il peut cependant en placer autant qu'il veut (nombre illimité). Apres la production et le dépot (qui forme une seule et même opération), les producteurs doivent signaler aux consommateurs la présence de pièces. Inversement, les consommateurs doivent aviser que les pièces ont été consommées et retirer de l'espace commun. Si l'espace est entièrement libre, n'importe le quel des deux producteurs peut y déposer une pièce. Les consommateurs peuvent accéder à l'espace en même temps que les producteurs, mais ne peuvent travailler sur la même pièce en même temps. Le producteur doit donc, au moins, devancer le consommateur du temps de production d'une pièce.

A) Ecrivez un moniteur et complétez les processus producteurs et consommateurs afin d'assurer la synchronisation de leurs activités.

B) Est-ce que la solution de A) donne lieu à des risques de famine (report à l'infini, 'starvation') de l'un des producteur? Si non, expliquez pourquoi. Si oui, suggérez une approche pour éviter les risques de famine, et modifiez (écrivez) le moniteur en conséquence.

SOLUTION:

A) Voir le programme qui suit. DebutPrX et FinCoX sont similaires au problème de la traversée de la rivière (traversée d'une pièce dans l'espace commun). FinPrX et DebutCoX s'assurent qu'une pièce produite sera consommée.

B) Il n'y a aucun risque de famine car il y a une priorité alternée entre les producteurs. Avant de produire une pièce, on vérifie si l'autre producteur est en attente, si oui on lui donne la priorité en se mettant en attente. En effet, quand toutes les pièces seront consommées, c'est l'autre producteur qui sera réactivé.

```
monitor  Mon
         export  DebutPrA, FinPrA, DebutPrB, FinPrB,
                 DebutCoA, FinCoA, DebutCoB, FinCoB
        var nPrA   : nat   := 0
        var nPrB   : nat   := 0
        var nCoA   : nat   := 0
        var nCoB   : nat   := 0
        var PrA, PrB, CoA, CoB : condition

      procedure DebutPrA
        if nPrB not= 0
          or not empty(PrB)
                then wait PrA end if
        nPrA := nPrA +1
      end DebutPrA

      procedure FinPrA
        nCoA := nCoA +1
        signal CoA
      end FinPrA

      procedure DebutCoA
        if nCoA = 0
                then wait CoA end if
        nCoA := nCoA -1
      end DebutCoA

      procedure FinCoA
        nPrA := nPrA -1
        if nPrA = 0
                then signal PrB end if
      end FinCoA
end Mon
```

```
      procedure DebutPrB
        if nPrA not= 0
          or not empty(PrA)
                then wait PrB end if
        nPrB := nPrB +1
      end DebutPrB

      procedure FinPrB
        nCoB := nCoB +1
        signal CoB
      end FinPrB

      procedure DebutCoB
        if nCoB = 0
                then wait CoB end if
        nCoB := nCoB -1
      end DebutCoB

      procedure FinCoB
        nPrB := nPrB -1
        if nPrB = 0
                then signal PrA end if
      end FinCoB
```

```
      process prodA
       loop
              Mon.DebutPrA
              { produit A }
              Mon.FinPrA
       end loop
      end prodA

      process prodB
       loop
              Mon.DebutPrB
              { produit B }
              Mon.FinPrB
       end loop
      end prodB
```

```
      process consA
       loop
              Mon.DebutCoA
              { consomme A }
              Mon.FinCoA
       endloop
      end consA

      process consB
       loop
              Mon.DebutCoB
              { consomme B }
              Mon.FinCoB
       endloop
      end consB
```

1.10.8. GESTION DES ACCÈS À UN DISQUE

SCHEDULING DISKS

Good scheduling algorithms can greatly improve the performance of operating systems. For example, the average turnaround time for jobs can be minimized by running short jobs before long jobs.

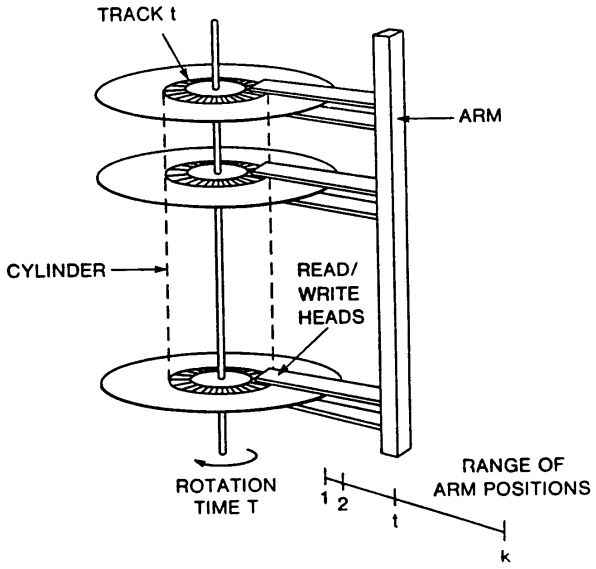
In this section, we want to minimize the time that processes wait for disk input and output. Before discussing disk scheduling algorithms, we need to know how disks access their data. A disk consists of a collection of platters, each with a top and bottom surface, attached to a central spindle and rotating at constant speed. There is usually a single arm, with a set of read/write heads, one per surface; the arm moves in or out, across the disk surfaces. This type of disk is called a movable head disk. When the arm is at a given position, the data passing under all read/write heads on all platters constitutes a cylinder. At a given cylinder position, the data passing under a particular read/write head constitutes a track.

Files of data are stored on a disk. A file consists of records; a record consists of fields of data. Some disks allow many records on a track. On other disks, a record must correspond exactly to a track. A program requests a data transfer to or from a disk by giving the cylinder number, track number, and record number.

There are various delay factors associated with accessing data on a movable head disk. Seek time is the time needed to position the arm at the required cylinder. Rotational delay is the time needed for the disk to rotate so that the desired record is under the read/write head. Transmission time is the time needed to transfer data between the disk and main memory.

Seek time increases with the number of cylinders that the arm is moved. Rotational delay can vary from zero to the time needed to complete one revolution; on the average, it is one-half the rotation time. Transmission time is dependent on the rotation time and recording density, and these factors vary from disk to disk. The following table shows that the seek time delay is the dominant factor in a typical transfer of a 1000-character record on a movable head disk.

Factor	Length of Time
Seek time	20.0 ms (milliseconds)
Rotational delay	5 ms
Transmission time	1 ms



Generally, a simple scheduling algorithm can do nothing to decrease rotational delay or transmission time. But ordering of disk read/write requests can decrease the average seek time. This can be accomplished by favoring those requests that do not require much arm motion.

Perhaps the simplest scheduling algorithm to implement is the FIFO algorithm; it moves the disk arm to the cylinder with the oldest pending request. With light disk traffic, the FIFO algorithm does a good job. But when the queue of disk requests starts building up, the simple nature of FIFO results in unnecessary arm motion. For example, suppose a sequence of disk requests arrives to read cylinders 12, 82, 12, 82, 12, and 82, in that order. (Such a sequence can easily occur if several jobs are using a file on cylinder 12 while others are using a file on cylinder 82.) The FIFO algorithm will unfortunately cause the disk to seek back and forth from cylinder 12 to 82 several times. A clever scheduling algorithm could minimize arm motion by servicing all the requests for cylinder 12 and

then all the requests for cylinder 82. We will discuss two scheduling algorithms that are more sophisticated than FIFO.

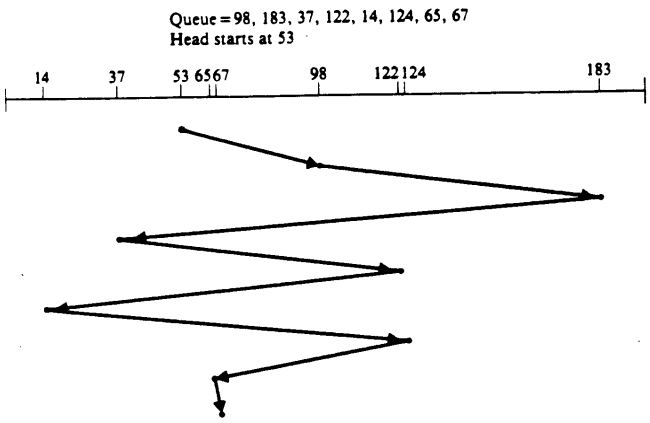


Figure 7.2 First-Come-First-Served disk scheduling

One such algorithm is the shortest seek time first (SSTF) algorithm. It operates as follows: the request chosen to be serviced next is the one that will move the disk arm the shortest distance from its current cylinder. The SSTF algorithm thus attempts to reduce disk arm motion. However, it can exhibit unwanted behavior; requests for certain cylinder regions on the disk may be indefinitely overtaken by requests for other cylinder regions closer to the current disk arm position. This results in certain disk requests not being serviced at all.

The SCAN algorithm is another scheduling algorithm that achieves short waiting times but prevents indefinite postponement. It attempts to reduce excessive disk arm motion and average waiting time by minimizing the frequency of change of direction of the disk arm. The SCAN algorithm operates as follows: while there remain requests in the current direction, the disk arm continues to move in that direction, servicing the request(s) at the nearest cylinder; if there are no pending requests in that direction (possibly because an edge of the disk surface has been encountered), the arm direction changes, and the disk arm begins its sweep across the surfaces in the opposite direction. This algorithm has also been called the elevator algorithm because of the analogy to the operation of an elevator, that runs up and down, receiving and discharging passengers.

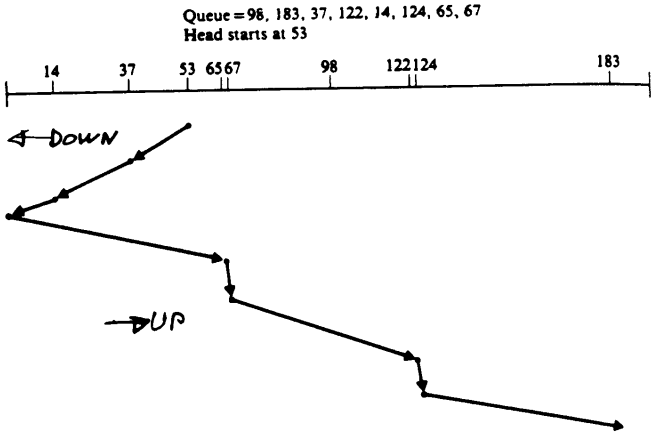


Figure 7.4 SCAN disk scheduling

A DISK ARM SCHEDULER

We will now give an implementation of the SCAN disk scheduling algorithm. SCAN gives good performance and illustrates the use of priority conditions. We will assume that there is a single disk and access to it is controlled by a monitor. The monitor has two entries:

Acquire(destCyl). Called by a process prior to transferring data to or from cylinder destCyl.

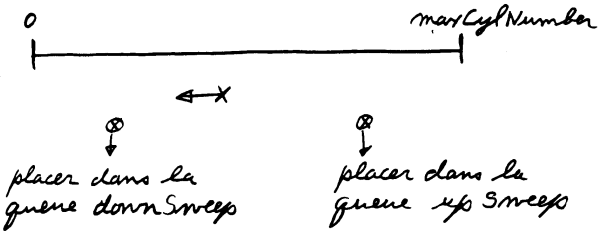
Release Called by a process after it has completed data transfer on the current cylinder.

The monitor must guarantee that only one process at a time uses the disk. The local data of the monitor keeps track of the current state of the disk. There must be a current disk arm position, armPosition (which can vary from 0 to the maximum cylinder number, maxCylNumber), the current direction of the arm sweep, direction (up or down), and a Boolean variable called inUse indicating whether the disk is currently busy. The up direction corresponds to increasing cylinder numbers, the down direction to decreasing cylinder numbers. A process using the disk executes the following:

Acquire (destination cylinder)
Read from or write to disk
Release

processus d'accès

[Scan.Acquire (destination)
déplace la tête de lecture
lecture ou écriture sur le disque
Scan.Release]



PRIORITY CONDITION

wait (upSweep , Priorité)
= destination

wait (downSweep , maxCylNumber - destination)

Here is the monitor implementing the SCAN algorithm:

```
 1 const maxCylNumber : 400

 2 var Scan: // Disk scheduling by SCAN (elevator)
                        algorithm
 3 monitor
 4   imports( maxCylNumber)
 5   exports( Acquire, Release)
 6   var armPosition: 0..maxCylNumber := 0
 7   var inUse: Boolean   := false
 8   pervasive const down := 0
 9   pervasive const up   := 1
10   var direction: down..up := up
11   var downSweep: priority condition
       // When not inUse and direction=down
12   var upSweep: priority condition
       // When not inUse and direction=up
13   procedure Acquire( destCyl:
                        0..maxCylNumber)
16     begin
17       if inUse then
18           if armPosition < destCyl or
                (armPosition=destCyl and
                        direction=down) then
19               wait( upSweep, destCyl)
20               assert( not inUse and
                        direction=up)
21           else
22               wait( downSweep,
                        maxCylNumber-destCyl)
23               assert( not inUse and
                        direction=down)
24           end if
25       end if
26       inUse := true
27       armPosition := destCyl
                        // Record arm position
28     end Acquire

29   procedure Release( )
32     begin
33       inUse := false
34       if direction = up then
35           if empty( upSweep) then
36               direction := down
37               signal( downSweep)
38           else
39               signal( upSweep)
40           end if
41       else
42           if empty( downSweep) then
43               direction := up
44               signal( upSweep)
45           else
46               signal( downSweep)
47           end if
48       end if
49     end Release

50 end monitor // Scan
```

It is particularly enlightening to discuss the SCAN monitor in detail. In the Acquire entry, if the disk is free the arm is moved to the desired cylinder and the process leaves the monitor. Otherwise, the process waits. It is not sufficient to use a single condition for waiting. (Why?) What we need is a set of conditions that relate the priority of a waiting request to the distance the requested cylinder is from the current cylinder. We could use an array of conditions, one for each cylinder. Then, when a process releases the disk, it would determine the "closest" non-empty condition waiting list in the current arm direction and signal that condition. This approach is clumsy and we will use priority conditions instead.

There are two priority conditions, each corresponding to a given arm direction (upSweep or downSweep). In the Acquire entry, a process waits on upSweep if its destination cylinder has a larger number than the current cylinder. A process waits on downSweep if its destination cylinder has a lower number than the current cylinder. Two questions arise: What happens if the destination cylinder equals the current cylinder? What priorities are specified in these waits?

We answer the priority question first. The priorities must indicate the distance the destination cylinder is from one end of the disk. For example, suppose the current arm position is at cylinder 25 and the direction is up. What happens with processes that request cylinders 100 and 200? Both requests are ahead of the disk arm and therefore will wait on upSweep. The request for cylinder 100 is closer than the request for cylinder 200. so cylinder 100 has priority over cylinder 200 (and therefore has a lower priority number). In our example, the request for cylinder 100 has a priority value of 100, while the one for cylinder 200 has a priority value of 200. Cylinder 100 will therefore be serviced before cylinder 200 on the upsweep.

Consider another example. Suppose the current cylinder is 250, and the direction is up. What happens with processes that request cylinders 150 and 50? Both requests are behind the disk arm and therefore will wait on downSweep. When the disk arm begins its sweep in the down direction, cylinder 150 is closer to it than cylinder 50. Assuming maxCylNumber is 400, cylinder 150 has priority over cylinder 50 (and should have a lower priority number). This can be accomplished by subtracting the destination cylinder from the maximum cylinder number to produce the priority value; the result will always be in the range from zero to maxCylNumber and will indicate the relative distance of arm motion. Cylinder 150 will have a priority value of 250 and cylinder 50 will have a priority value of 350. Cylinder 150 will therefore be serviced before cylinder 50 on the down sweep.

In our example, the priority values are specified in the wait statements in lines 19 and 22. We have been careful in our discussion here to be explicit about the arm direction, because the correct condition to wait on and the correct priority depend on that direction. But the arm direction is only tested (in line 18) when the destination cylinder equals the current cylinder.

The answer to this apparent mystery is that the arm direction generally does not matter. If the arm position is less than the destination cylinder, the process should wait on upSweep regardless of the arm direction. If the arm position is greater than the destination cylinder, the process should wait on downSweep regardless of the arm direction. Only when the arm position equals the destination cylinder does the arm direction matter. We are back to the first question we asked, so it is appropriate to answer it now.

To see what happens if the destination cylinder equals the current cylinder, we will discuss an alternative to line 18, showing that it can lead to indefinite postponement. (Questions 10 and 11 in the Exercises mention other alternatives.)

Consider the following alternative to line 18:

```
if armPosition < destCyl then
```

A process with a destination cylinder equal to the current cylinder will wait on downSweep in line 22. Suppose the current direction of the disk is down. A process releasing the disk will signal downSweep in line 46 because there is a process waiting on it. If there is a stream of processes similar to the first one in this example, all with requests for thc current cylinder, the disk arm will remain at the current cylinder servicing all these requests. Indefinite postponement results because

requests for other cylinders are being continually denied.

The alternatives here and in the Exercises show that implementing the SCAN algorithm is a tricky matter. How does the current line 18 preclude indefinite postponement?

```
if armPosition < destCyl or
   (armPosition = destCyl and direction = down)
   then
```

A process with a destination cylinder equal to the current cylinder will wait on upSweep when the direction is down. When the direction is up, such a process will wait on downSweep. Thus, in the original line 18, a process with a destination cylinder equal to the current cylinder waits on the condition variable associated with the opposite of the current arm direction. Such a request does not cause the disk arm to remain at the current cylinder because it does not wait on the condition associated with that direction. The request will not be serviced on the current sweep, but on the next sweep in the opposite direction. Indefinite postponement is precluded in this way.

There is another benefit in the way we organized our solution. All requests waiting for a cylinder before the arm gets to that cylinder will be served on the same sweep. This benefit comes from the priority values. All requests for the same cylinder have the same priority value. When the destination cylinder equals the current cylinder, the priority value associated with the destination cylinder is the smallest priority value in the current arm direction. Thus, the signals in lines 39 and 46 will resume all processes that requested the current cylinder and were waiting before the arm got to that cylinder, before going on to another cylinder in the same direction or changing directions. This improves performance because it reduces waiting time.

In the Release entry, control of the disk is returned and the process to be serviced next is signaled. If there is a waiting request in the current direction, it is signaled (lines 39 and 46). If there is no waiting request in the current direction, the arm direction is changed (lines 36 and 43) and a waiting request in the new direction is signaled (lines 37 and 44). In all cases, the process to be serviced next is the one that has the closest request in the current arm direction.

We have devoted much space to considerations of indefinite postponement. Deadlock is a simpler matter.

Deadlock cannot occur in our solution. Processes are never blocked in their process code. In the monitor code, only a single resource (the disk) is being used, processes can never hold some resource units while requesting others. Finally, processes wait on the correct conditions and specify the correct priority values.

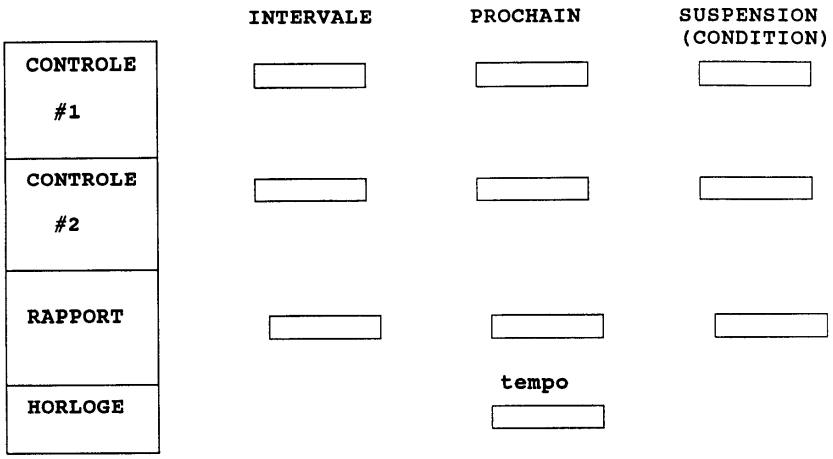
1.10.9. ORDONNANCEUR TEMPS RÉEL

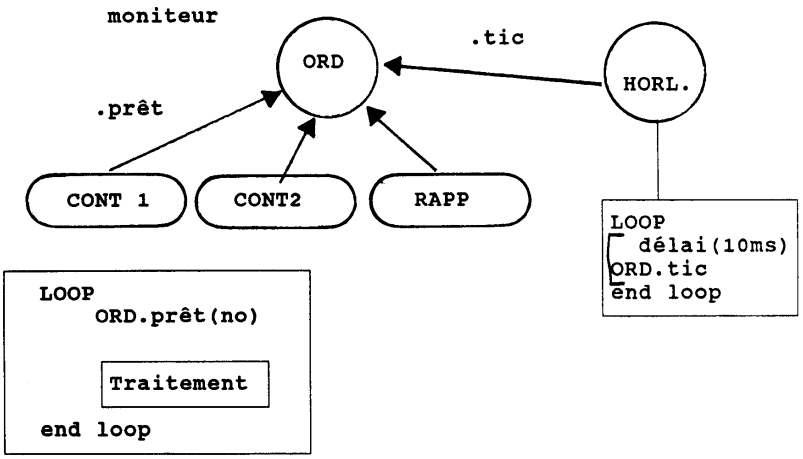
Nous examinons un problème d'ordonnancement temps-réel ou nous avons plusieurs processus qui doivent être exécutés à intervalle régulier. Cela peut être le cas, par exemple, pour des problèmes de contrôle dans une usine où à l'intérieur d'une même machine. Un processus qui contrôle un appareil doit tourner constamment (doit toujours s'exécuter) pour éviter que l'appareil hors contrôle n'effectue des mouvements dangereux. Contrôler une machine implique que le processus vérifie à intervalle régulier l'état de la machine et apporte des correctifs au besoin. On utilise souvent une boucle d'asservissement. On précise au processus l'objectif à atteindre. Le processus vérifie à intervalle régulier l'état de la machine et calcul les commandes qu'il doit donner à la machine pour la diriger vers l'objectif. Si un ordinateur contrôle plusieurs machines, il doit partager son temps entre les processus de contrôle de chacune des machines. Il faut cependant s'assurer que chaque processus puisse exécuter une itération de sa boucle d'asservissement à intervalle régulier.

Comme exemple, considérons une section d'une usine ou l'opérateur dispose d'une console (ordinateur) pour contrôler deux appareils qui assurent le contrôle et la surveillance d'un procédé. On a donc deux processus de contrôle qui doivent être exécutés à toutes les secondes. De plus, nous avons un processus qui doit afficher sur l'écran de l'opérateur l'état du système. Ce processus *Rapport* doit donc mettre à jour l'écran à toutes les 30 secondes. Nous utilisons un moniteur *ORD* pour gérer l'ordonnancement des processus. Le moniteur doit maintenir une horloge. Il contient une variable *temps* indiquant l'heure en unité de 10 millisecondes. Il faut donc ajouter un processus *Horloge* qui va appeler la procédure *tic* du moniteur à toutes les 10 millisecondes. Le moniteur dispose également d'un vecteur *intervalle* indiquant à quelle intervalle chacun des processus doit être exécuté.

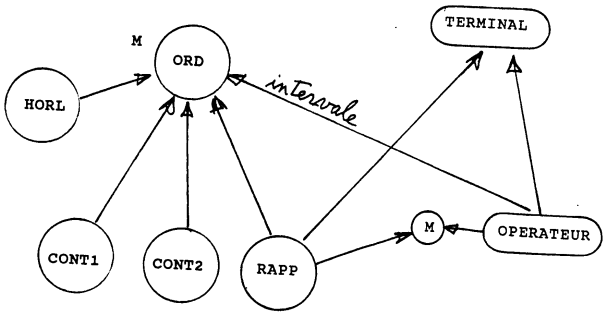
Les processus *Contrôle-1*, *Contrôle-2* et *Rapport* sont constitués d'une boucle contenant les instructions à effectuer. Au début de la boucle, on appelle la procédure *prêt* du moniteur pour suspendre les processus. Chaque processus sera réveillé par le moniteur lorsque le délai prévu (intervalle) sera écoulé. On considère que le temps d'exécution des instructions dans la boucle est petit comparé à la valeur de l'intervalle de réveil. Le moniteur dispose d'un vecteur de condition *suspension* pour suspendre chacun des processus. Un vecteur *prochain* indique à quel moment un processus doit être réveillé. La procédure *tic*, appelée à chaque 10 millisecondes, incrémente le temps du moniteur et vérifie si le temps de réveil d'un processus est atteint. Si oui, elle détermine le prochain temps de réveil en incrémentant *prochain* de la valeur de *intervalle*, et réveille le processus. La procédure *prêt* suspend le processus sur la condition *suspension(no)*, *no* correspondant au numéro du processus.

Nous pouvons enrichir notre système de contrôle en ajoutant un processus *Opérateur* qui reçoit les commandes de l'opérateur par le clavier. Nous pouvons aussi ajouter une procédure au moniteur *ORD* pour permettre au processus *Opérateur* de modifier les valeurs des intervalles. Ce processus doit également utiliser le terminale (l'écran de l'opérateur) pour répondre à ses commandes. Il doit partager l'écran avec le processus *Rapport*. Un moniteur supplémentaire peut alors être utilisé pour assurer l'exclusion mutuelle entre ces deux processus et garantir le bon fonctionnement de l'écran.






```
monitor ORD
    exports pret, tic
    var intervalle:  array 1 .. 3 of nat
    var prochain:    array 1 .. 3 of nat
    var suspension:  array 1 .. 3 of condition
    var temps:       nat := 0

    procedure pret ( no: nat )
        wait suspension( no )
    end pret

    procedure tic
        temps += 1
        for no : 1 .. 3
            if prochain( no ) <= temps then
                prochain( no ) += intervalle( no )
                signal suspension( no )
            end if
        end for
    end tic
end ORD
```



1.11. Moniteur dans Turing Plus

1.11.1. Processus concurrent

Un programme comporte normalement un seul trait d'exécution correspondant au programme principal. Les procédures ne sont exécutées que lorsqu'elles sont appelées par le programme principal. On peut définir des processus qui s'exécuteront en concurrence avec le programme principal. Un programme termine seulement lorsque tous les processus et le programme principal ont terminé.

process définit un processus et possède la même syntaxe que la définition de procédure. On remplace le mot clé procedure par process. On peut donc passer des paramètres à un processus.

fork *processus* Il faut faire un fork pour amorcer l'exécution d'un processus. La définition d'un processus n'est pas suffisante pour son exécution.

Le programme suivant crée deux processus qui impriment de manière non déterministe et entrelacée les chaînes de caractères "Hi" et "Ho". Il y a deux fork créant deux processus auxquels on a passé des paramètres différents.

définition des processus

```
process speak (word: string)
    loop
        put word
    end loop
end speak
```

programme principale

```
fork speak( "Hi " )
    // Start saying: Hi Hi Hi
fork speak( "Ho " )
    // Start saying: Ho Ho Ho
```

Les deux processus vont s'exécuter en parallèle à une vitesse relative (d'un par rapport à l'autre) non déterminée. La sortie du programme est une séquence non prévisible de "Hi" et "Ho" tel que

Ho Ho Hi Ho Hi Hi Hi Hi Ho

L'ordre de "Hi" et "Ho" peut varier d'une exécution à l'autre. Il est même possible qu'une partie de la chaîne "Hi" (seulement le "H") soit imprimée, suivi du "Ho" et finalement du reste de la chaîne "Hi". L'impression peut donc être affectée par la manière que le put est implanté.

En Turing Plus, les procédures et fonctions sont ré-entrantes (procédure récursive). Ceci permet à deux processus d'appeler la même procédure. Une procédure ré-entrante est obtenue en utilisant la pile (stack) pour contenir les paramètres et les variables locales de la procédure. Nous pouvons donc réécrire le programme Hi-Ho comme suit:

```
procedure Speak (word: string)
    loop
        put word
    end loop
end Speak

process Hi
    Speak( "Hi " )
end Hi

process Ho
    Speak( "Ho " )
end Ho
fork Hi      // Start saying: Hi Hi Hi
fork Ho      // Start saying: Ho Ho Ho
```

1.11.2. Moniteur

Un moniteur est un module adapté pour la programmation concurrente. La définition d'un moniteur suit la même forme (syntaxe) que la définition d'un module. On remplace seulement le mot de module par le mot clé monitor dans l'entête (voir module). Le moniteur garantit l'exclusion mutuelle des processus qui exécutent une procédure du moniteur. Un seul processus peut exécuter une procédure du moniteur à la fois. L'exclusion mutuelle est un mécanisme utile pour assurer l'intégrité des données du moniteur.

Le programme suivant comporte deux processus: Observer qui compte toutes les observations qu'il

fait, et Reporter qui écrit un rapport sur le nombre d'observations effectuées. Un moniteur Update est utilisé pour assurer l'intégrité des données partagées par les deux processus concurrents.

```
monitor Update
    export Observe, Report
    var count: nat := O

    procedure Observe
        count := count + i
    end Observe

    procedure Report
        put count
        count := O
    end Report
end Update

process Observer
    loop
        wait for an event
        Update.Observe
    end loop
end Observer

process Reporter
    loop
        wait for a while
        Update.Report
    end loop
end Reporter

fork Observer
fork Reporter
```

La donnée du moniteur est la variable count. Le moniteur est initialisé avant la création des processus par l'instruction fork, alors la variable count débute avec une valeur de zéro. Le processus Observer appelle l'entrée Update. Observe, qui incrémente count d'un. Le processus Reporter appelle Update.Report pour imprimer la valeur de count et la remettre à zéro. Si le Reporter essaie d'entrer dans le moniteur tandis qu'Observer exécute dans le moniteur, le Reporter est bloqué au point d'entrée jusqu'à ce que l'Observer quitte le moniteur. De même, l'Observer est empêché de rentrer dans le moniteur quand le Reporter est à l'intérieur.

Les données du moniteur, la variable count dans cet exemple, est statique et demeure intacte entre les invocations du moniteur. Le moniteur, tout comme le module, peut avoir du code avec la déclaration des données pour les initialiser.

1.11.3. Condition, wait et signal

A l'intérieur d'un moniteur seulement, on peut définir des variables de type condition. Une condition est une queue de processus contenant les processus qui sont suspendu sur la condition. Une condition peut être utilisée avec les instructions wait et signal.

wait condition

Cette instruction suspend le processus qui l'exécute et le place sur la queue de la condition. Un processus suspendu ne s'exécute plus dans le moniteur et on considère qu'il l'a quitté.

signal condition

Lorsqu'un processus exécute l'instruction signal, on vérifie la queue de la condition.

1) Si la queue contient des processus, un des processus est retiré de la queue et on commence à l'exécuter immédiatement. Le processus qui à fait le signal quitte alors le moniteur. Il pourra continuer seulement lorsqu'il n'y aura plus de processus dans le moniteur.

2) Si la queue contient aucun processus, alors aucun processus n'est réveillé et le processus avec l'instruction signal continue son exécution.

empty (condition)

empty est une fonction qui retourne true si la queue de la condition est vide, i.e. s'il n'y a pas de processus suspendu sur la condition. Si non, elle retourne false.

var c: **condition priority**
wait c, **13**

On peut ainsi déclarer une condition avec priorité. L'instruction wait doit alors indiquer la priorité, par exemple, "wait c, 13 ". C'est le processus avec la plus basse priorité qui sera réveillé le premier par une instruction signal. La priorité est un entier positif.

var c: **condition deferred**

On peut ainsi déclarer une condition différée. Un processus effectuant un signal sur cette condition continue son exécution, le processus 'réveillé' sera exécuté après (en différé).

Notez qu'un seul processus à la fois est présent dans le moniteur, assurant un accès en exclusion mutuelle aux données du moniteur.

Une condition ne peut pas servir pour la définition dc type, ne peut être utilisée dans un record, ne peut être déclarée dans une instruction (tel que loop et begin) ou dans une sous-routine. Elle peut seulement être déclarée dans un moniteur.

Il n'y a aucun ordre de progression assuré parmi les processus suivants: 1) les processus réveillés sur une condition différée, 2) les processus exécutant une instruction signal sur une condition non-différée, et 3) les processus voulant exécuter une procédure (ou fonction) du moniteur. On ne sait donc pas lequel parmi ces processus va s'exécuter en premier (dans le moniteur).

Un moniteur peut être utilisé pour gérer l'allocation des ressources communes entre plusieurs processus. Quand une ressource est allouée à un processus, les autres processus qui demandent la ressource doivent être bloqués jusqu'à ce que la ressource soit libérée. Le moniteur suivant gère l'allocation d'une seule ressource partagée par plusieurs processus. Chaque processus doit d'abord "acquérir" la ressource, puis l'utiliser et finalement la relâcher. Dans la procédure acquire, le processus se suspend en exécutant l'instruction wait si la ressource n'est pas disponible. Dans la procédure Release, l'instruction signal réveille un des processus en attente (s'il y en a un). S'il n'y a pas de processus en attente, le processus exécutant l'instruction signal continue.

```
monitor Resource
    export Acquire, Release

    var inUse: boolcan := false
    var available: condition  // Signal if not mUse

    procedure Acquire
        if inUse then
            wait available
        end if
        inUse := truc
    end Acquire

    procedure Release
        inUse := false
        signal available
    end Release
end Resource
```

Les procédures Acquire et Release du moniteur correspondent aux opérations P-wait et V-signal sur un sémaphore binaire. Ceci montre que l'on peut implanter des sémaphores à l'aide de moniteur. L'inverse est également possible, quoique plus difficile. Les sémaphores et les moniteurs ont donc la même puissance.

1.11.4. Pause

```
pause 10    // pause de 10 unité de temps
```

L'instruction pause permet de suspendre un processus pour une durée spécifiée par le paramètre. Il s'agit d'un temps simulé. Turing Plus en fait simule l'exécution de processus concurrents. De même, il peut gérer une horloge virtuelle et suspendre les processus pour un certain nombre d'unité de temps virtuel.

Au début du programme, l'horloge est mise à zéro. Par la suite, seulement des instructions pause peut la faire avancer. C'est-à-dire que toute autre instruction est considérée prendre un temps nul. Au besoin, le programmeur peut indiquer qu'une partie de code prend un certain nombre d'unité de temps en insérant une instruction pause dans le code. Lorsqu'un processus exécute une instruction pause, Turing Plus mémorise le temps de réveil en ajoutant la valeur du délai au temps actuel de l'horloge. On exécute alors tous les processus qui ne sont pas suspendus sur une condition ou sur une instruction pause. Lorsqu'aucun processus ne peut plus être exécuté, alors on avance l'horloge. On choisit le processus suspendu sur une instruction pause ayant le temps de réveil le plus petit, on avance l'horloge à ce temps de réveil et on exécute ce processus. Le réveil de ce processus peut amener le réveil et l'exécution d'autres processus. L'horloge sera de nouveau avancée lorsqu'il n'y aura plus de processus exécutable.