

Sémaphore

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

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

1) Notion de sémaphore

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

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

a) Fonctionnement de P et V

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

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

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

```

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

```

Remarque: Il est possible de considérer le compteur d'un sémaphore comme ayant toujours une valeur entière positive. Ainsi, cette valeur peut représenter le nombre de points d'accès à une ressource; ce nombre, bien entendu, ne peut être que positif. Toutefois, les algorithmes des primitives P et V doivent correspondre à cette nouvelle définition:

```

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

```

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

Bien que par abus de langage ou pour simplicité, on désigne un sémaphore par un entier seulement, l'implémentation réelle dans un système impose d'associer à cet entier un pointeur vers sa file d'attente associée.

Dans la suite, sauf indication contraire, on adoptera la première définition. On peut utiliser la structure d'implémentation suivante:

```

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

```

b) Caractéristiques de P et V

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

La stratégie selon laquelle est gérée la file Qsem est déterminée lors de l'implémentation. Elle doit être choisie suivant l'objectif pour lequel le sémaphore correspondant est utilisé.

En général, le noyau peut adopter la stratégie FCFS (First-Come First-Served) pour sortir de Qsem le processus ayant le plus longtemps attendu (stratégie équitable). Il peut également choisir un algorithme plus complexe tel celui basé sur la priorité des processus lorsque le temps d'attente dans Qsem devient significatif.

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

c) Utilisation de sémaphores

Dans un cas tout à fait général, un sémaphore de valeur initiale quelconque positive ou nulle (par exemple N) peut être utilisé pour synchroniser l'accès des processus concurrents à une ressource partageable à N points d'accès. Cette ressource peut donc être allouée simultanément à N processus. Le sémaphore est alors considéré comme représentant le nombre de copies de la ressource.

Lorsqu'une ressource commune R n'a qu'un seul point d'accès ($N=1$:un seul exemplaire) autrement dit ressource critique, l'usage d'un sémaphore à valeur initiale unitaire permet de réaliser l'exclusion mutuelle lors des accès à R.

Donc pour réaliser l'exclusivité d'exécution d'une section de programme : I.e. créer des sections critiques, il y a lieu de faire usage de sémaphores à valeur initiale égale à 1. Ces sémaphores portent le nom de sémaphore d'exclusion mutuelle. Aussi, toute section de programme deviendra *section critique* en l'encadrant par $P(mutex)$ à l'entrée de la section (prologue), et $V(mutex)$ à la sortie (épilogue) avec bien entendu mutex: sémaphore initialisé à 1.

Exemple : *Var* mutex: semaphore;

mutex := 1;

P(mutex) % prologue %

< section critique >;

V(mutex)

% épilogue %

Lorsque la valeur initiale d'un sémaphore est nulle (0), celui-ci est désigné par *sémaphore privé*. Il permet, en particulier, au processus détenteur de ce sémaphore de se bloquer en attente d'un événement particulier. Cet événement peut être: une communication synchrone de message, une attente de la fin d'une entrée/sortie, etc. La gestion de la file d'attente de ce type de sémaphore est très simple puisqu'elle ne peut contenir que le processus détenteur du sémaphore.

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

Var semprive1: semaphore init 0;

semprive2: semaphore init 0;

P1

P2

P(semprive1);

Recevoir (P2, msg);

Emettre (P1, msg);

V(semprive1);

Emettre (P2, msg);

V(semprive2);

P(semprive2);

Recevoir (P1, msg);

Avec Emettre (Pi, msg): Transmettre un message *msg* au processus Pi,

Recevoir(Pj, msg): Recevoir du processus Pj un message *msg*.

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

La même situation se produit lorsqu'un processus effectue une E/S synchrone. L'attente de cette E/S peut se faire par l'intermédiaire de l'exécution d'une opération P sur un sémaphore privé. Le réveil peut être réalisé par l'exécution de l'opération V sur ce même sémaphore privé, dans la routine d'interruption associée à la fin de l'E/S.

Un sémaphore dont la valeur passe alternativement de 0 à 1 et réciproquement porte le nom de sémaphore *binnaire*.

L'outil sémaphore est intrinsèquement général et complet mais conduit souvent à une structuration de programme difficile à comprendre (peu lisible) et sensible à des modifications. Autrement dit, la dispersion des primitives P et V à travers le code d'un programme, particulièrement lorsque celui-ci est de taille non négligeable et comprenant un nombre important de points de synchronisation, rend sa lecture (compréhension) fastidieuse. De même, une modification (ajout ou retrait) d'une partie de code renfermant un ou plusieurs

points de synchronisation peut mettre en cause la validité de la partie contrôle (synchronisation) du programme tout entier!

L'inconvénient premier se situe notamment au niveau de la programmation qui requiert du programmeur une attention particulière; en l'occurrence, un oubli d'une primitive V (correspondante à P) ou l'écriture de P à la place de V conduit directement à un blocage infini.

d) Implémentation des primitives P et V

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

2) Problèmes possibles de l'exclusion mutuelle

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

a) L'Interblocage

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

Cet état résulte du fait que chacun des processus (en concurrence) est en possession d'une ou plusieurs ressources réclamée(s) par d'autres et réciproquement. En d'autres termes, les processus concurrents se trouvent bloqués par manque de ressources, et l'événement de déblocage ne peut provenir que de l'un d'entre eux (déjà bloqué) ou par intervention extérieure forcée (système d'exploitation, opérateur humain).

Plus précisément, un interblocage de processus concurrents se produit dans un système si et seulement les quatre conditions suivantes se réalisent conjointement, à savoir:

- 1- *Exclusion mutuelle*: Demandes concurrentes d'utilisation de ressource(s) non partageable(s). Autrement dit, il existe au moins une ressource demandée dont l'utilisation doit être en exclusion mutuelle,
- 2- *Allocation et attente*: Un des processus concurrents est en possession d'au moins une ressource non partageable et attend l'allocation d'autres ressources détenues par d'autres processus,

- 3- *Attente circulaire*: Il existe dans le système un ensemble de processus en attente $\{P_0, P_1, \dots, P_n\}$ de sorte que P_0 attende une ressource détenue par P_1 , P_1 attend une ressource en possession de P_2 , ... , et P_{n-1} attend une ressource possédée par P_n lequel attend une ressource détenue par P_0 .
- 4- *Absence de réquisition*: Une ressource allouée à un processus P_i , ne peut être réquisitionnée avant sa libération naturelle (à la fin de l'exécution normale de P_i).

Remarque: En général, assurer la sécurité d'un système revient à identifier les problèmes indésirables pouvant l'affecter, et prendre les mesures appropriées pour qu'ils ne puissent survenir, ou à défaut, être en mesure de confiner ou minimiser les dégâts si jamais ceux-la se produisent. Ainsi, étant un problème indésirable, l'interblocage est théoriquement considéré comme une propriété de sécurité (de la sûreté de fonctionnement) que tout système valide doit en être exempté.

Exemple:

Soient deux processus P_1 et P_2 demandant l'allocation de deux types de ressources non partageables d'une installation, à savoir imprimantes et bandes magnétiques. En admettant que ces ressources existent chacune en exemplaire unique, le scénario d'exécution de P_1 et P_2 suivant conduit inévitablement à un interblocage (les quatre conditions précédentes sont simultanément vérifiées).

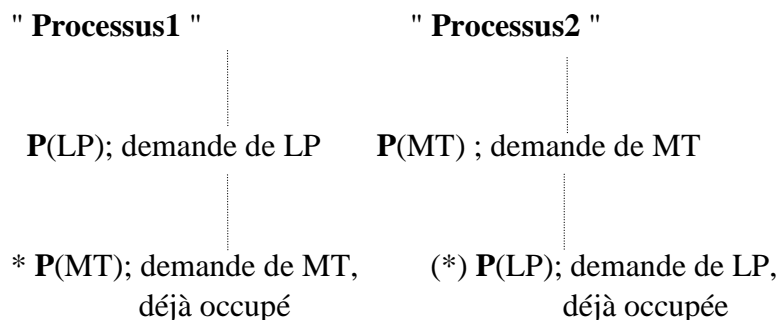


Figure 4. Exemple d'interblocage

(*): à ce stade, les demandes ne peuvent être satisfaites; P_1 bloque P_2 qui à son tour bloque P_1 , c'est l'inévitable interblocage.

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

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

- Ces dommages peuvent se limiter à un simple préjudice au niveau temporel, c'est-à-dire: l'application sera de nouveau lancée et les résultats attendus seront produits avec un retard acceptable. Il faut noter que ce retard peut ne pas être acceptable, en particulier dans des applications dites *temps réel*, où le facteur temps est aussi important que les résultats corrects fournis. Autrement dit, les résultats attendus sont acceptables s'ils sont corrects et délivrés dans les délais impartis.

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

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

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

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

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

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

b) Privation

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

Pratiquement, la différence entre un interblocage et une privation peut être d'ordre temporel ou dynamique. Dans le premier cas, l'évolution de l'ensemble des processus bloqués est compromise tant qu'il n'y ait pas d'intervention extérieure au processus; alors que dans le second, un ou plusieurs processus monopolisent le contrôle du processeur au détriment d'un ou plusieurs autres processus qui peuvent rester bloqués indéfiniment. Logiquement, le nombre de processus est fini, et un processus arrive au terme de son exécution au bout d'un temps fini (même dans trente six heures!); donc, tôt ou tard le ou les processus en privation

obtiendront le contrôle du processeur. La pénalité n'est donc que temporelle (pour une catégorie de processus), mais elle peut s'avérer grave dans un environnement qui ne peut supporter une telle contrainte (exemple: systèmes temps réels).

L'intervention précitée consiste en général à rompre le *cycle de blocage* mutuel par "destruction" délibérée d'un des processus impliqués. Ce cycle de blocage est déterminé à partir du graphe de dépendance des processus construit par le système au fur et à mesure des demandes d'allocation de ressources (voir plus loin).

Deux stratégies peuvent être appliquées quant au choix du processus victime de la destruction.

- La première désigne comme victime le processus ayant accaparé le maximum de ressources; de cette manière, la libération de ces ressources permettrait l'évolution d'un nombre appréciable de processus bloqués. A noter que si le processus cible se trouve dans la phase terminale de son exécution, il subirait un préjudice certain. Cette stratégie qui a tendance à favoriser le degré de multiprogrammation s'avère intéressante dans les systèmes temps partagé.

- La deuxième stratégie considère comme cible de destruction le processus bloqué le plus jeune possible. On espère ainsi pénaliser le moins possible la victime (le traitement à refaire est minimum). On peut considérer que le processus victime de la première stratégie est le moins jeune si on estime que le nombre de ressources obtenues est une fonction croissante du temps d'exécution.

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

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