Cours Programmation Système - Sémaphore

Thierry Vaira

BTS IRIS Avignon

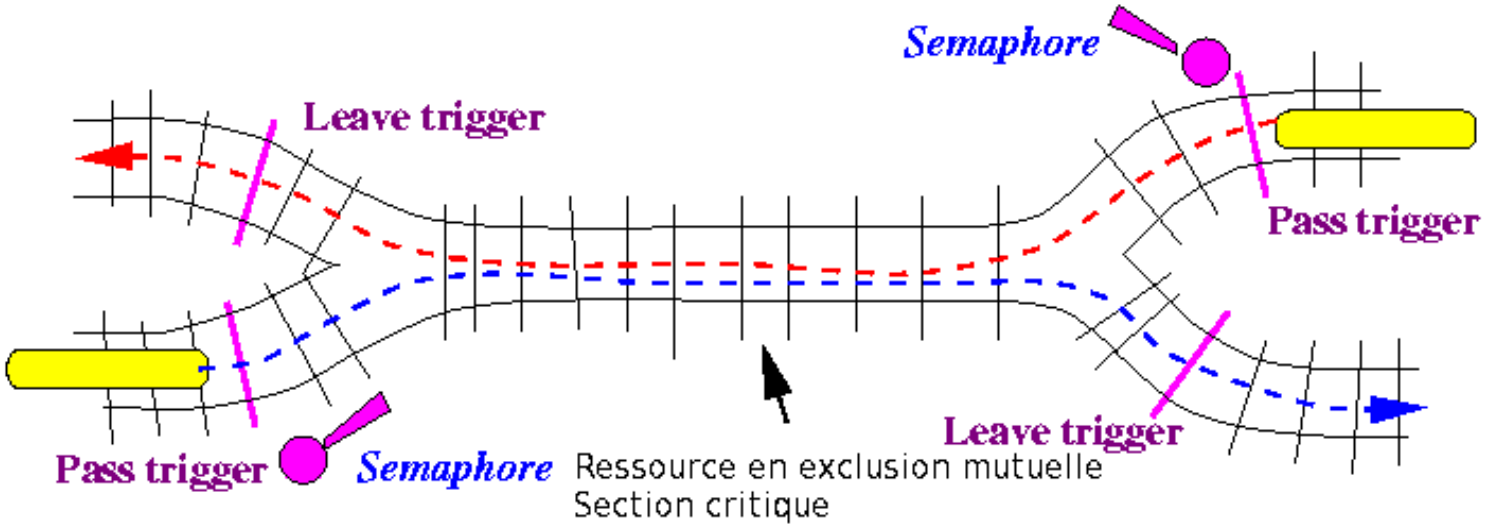
tvaira@free.fr ◎ v0.1

Sommaire

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Objectifs

- Découvrir les sémaphores et leur utilisation.
- Être capable de mettre en oeuvre des sémaphores dans un environnement de programmation concurrente.



Le sémaphore a été inventé par Edsger Dijkstra et utilisé pour la première fois dans le système d'exploitation *THE Operating system*, créé dans les années 1960.

Définitions

- Un **sémaphore** est une **variable** et constitue la méthode utilisée couramment pour **restreindre l'accès à des ressources partagées dans un environnement de programmation concurrente**.
- Une **section critique** est une partie de code telle que 2 processus ne peuvent s'y trouver au même instant.
- Une ressource est en **exclusion mutuelle** si seul un processus peut utiliser la ressource à un instant donné.

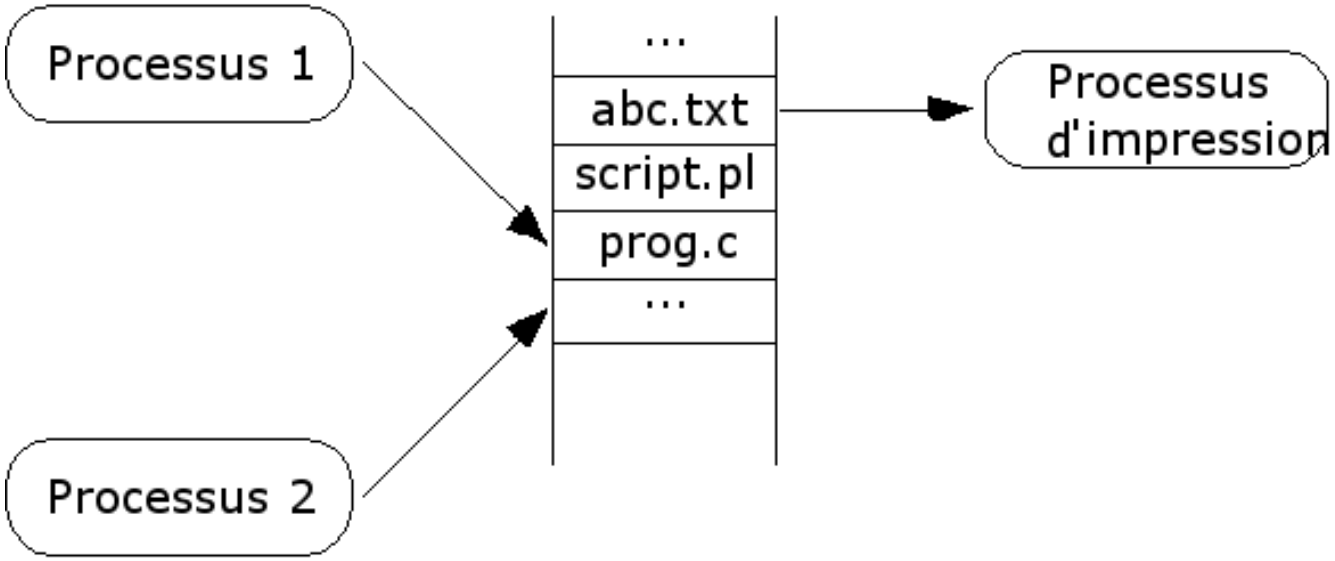
Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Exemple introductif

On considère le fonctionnement d'un *spool* d'impression. La file d'impression est gérée à l'aide de 2 variables partagées `in` et `out` :

- `in` → pointe sur le premier emplacement libre
- `out` → pointe sur le prochain fichier à imprimer

Les processus désirant imprimer un document doivent le déposer dans la file comme ceci : `file[in] = fichier ; in++ ;`



Mise en évidence du problème

Dans un environnement multi-tâche, l'ordre d'exécution des instructions n'étant pas garanti, il est possible d'obtenir le déroulement suivant :

temps	P1	P2
(1)	file[in] := "F1";	
(2)		file[in] := "F2";
(3)		in++;
(4)	in++;	

Problème : l'impression du fichier n° 1 (F1) déposé par P1 ne se fera pas (car il a été "écrasé" par F2) et l'état de la file est incohérente (car il y a une double incrémentation de `in`) !

Solution : aucune commutation de processus ne doit s'effectuer entre la dépose du nom du fichier et l'incrémentation du pointeur `in`. Une solution consiste à interdire à un processus d'exécuter la procédure impression si un autre processus non actif n'a pas fini l'exécution de celle-ci.

Les sémaphores

- Un **sémaphore S** est **une variable entière** qui n'est accessible qu'au travers de 3 opérations **Init**, **P** et **V**.
- La **valeur** d'un sémaphore est le **nombre d'unités de ressource libres**.
- Un sémaphore initialisé à 1 est un **sémaphore binaire** et permet de contrôler l'accès à une seule ressource.
- Les opérations **P** et **V** doivent être indivisibles (atomiques), ce qui signifie qu'elles ne peuvent pas être exécutées plusieurs fois de manière concurrente. Un processus qui désire exécuter une opération qui est déjà en cours d'exécution par un autre processus doit attendre que le premier termine.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

L'opération Init

- `Init(S, valeur)` est seulement utilisé pour **initialiser le sémaphore S avec une valeur**.
- Cette opération ne doit être réalisée qu'une seule et unique fois.

L'opération P

- L'opération `P(S)` (du néerlandais *Proberen* signifiant tester et en français "Puis-je ?" ou éventuellement Prise ou Prendre) est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant.
- L'opération `P(S)` réalise les instructions suivantes de manière atomique :

```
SI (S > 0)
   ALORS S--;
   SINON (attendre sur S)
FSI
```

L'opération V

- L'opération `V(S)` (du néerlandais *Verhogen* signifient tester incrémenter et en français "Vas-y !" ou éventuellement Vente ou Vendre) est l'opération inverse. Elle rend simplement une ressource disponible à nouveau après que le processus a terminé de l'utiliser.
- L'opération `V(S)` réalise les instructions suivantes de manière atomique :

```
S++;
SI (des processus sont en attente sur S)
   ALORS laisser l'un deux continuer (reveil)
FSI
```

Le sémaphore binaire

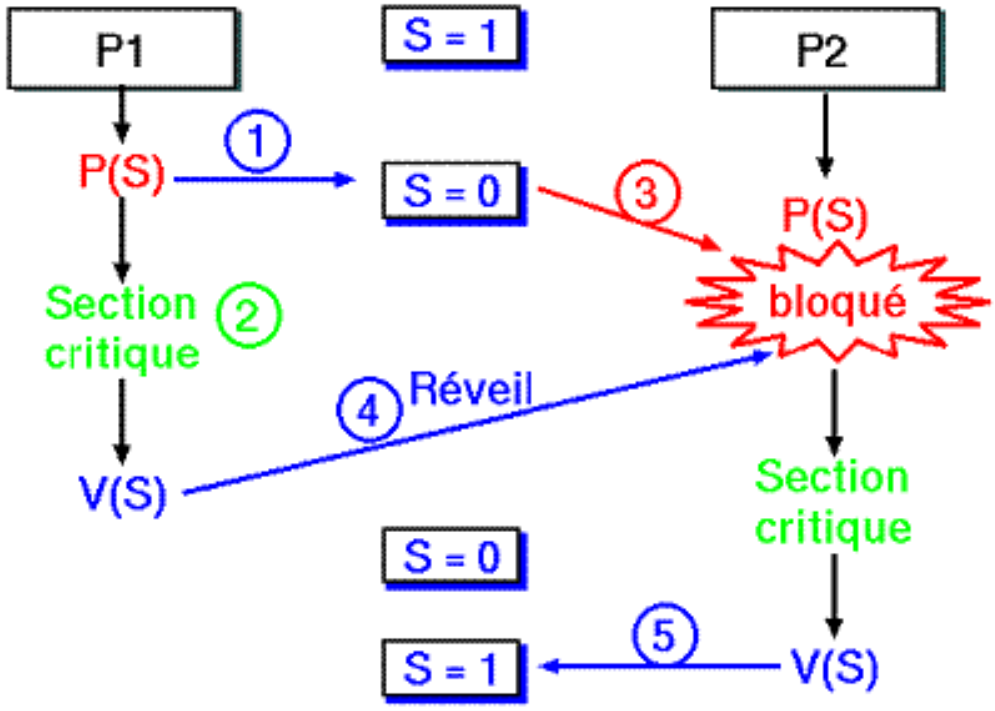
- Un **sémaphore binaire** est un **sémaphore qui est initialisé avec la valeur 1**.
- Ceci a pour effet de contrôler l'accès une ressource unique.
- Le sémaphore binaire permet l'**exclusion mutuelle (*mutex*)** : une ressource est en exclusion mutuelle si seul un processus peut utiliser la ressource à un instant donné.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Accès à une section critique

L'accès à la section critique en utilisant un sémaphore binaire sera alors :

```
P(S);
SectionCritique;
V(S);
```



Exemple d'interblocage

Exemple d'interblocage avec deux *mutex* (nommés M1 et M2) :

```
TâcheA :
  Obtenir M1
  Obtenir M2
  Action nécessitant les deux verrous
  Rendre M2
  Rendre M1
```

```
TâcheB :
  Obtenir M2
  Obtenir M1
  Action nécessitant les deux verrous
  Rendre M1
  Rendre M2
```

Un interblocage est possible si :

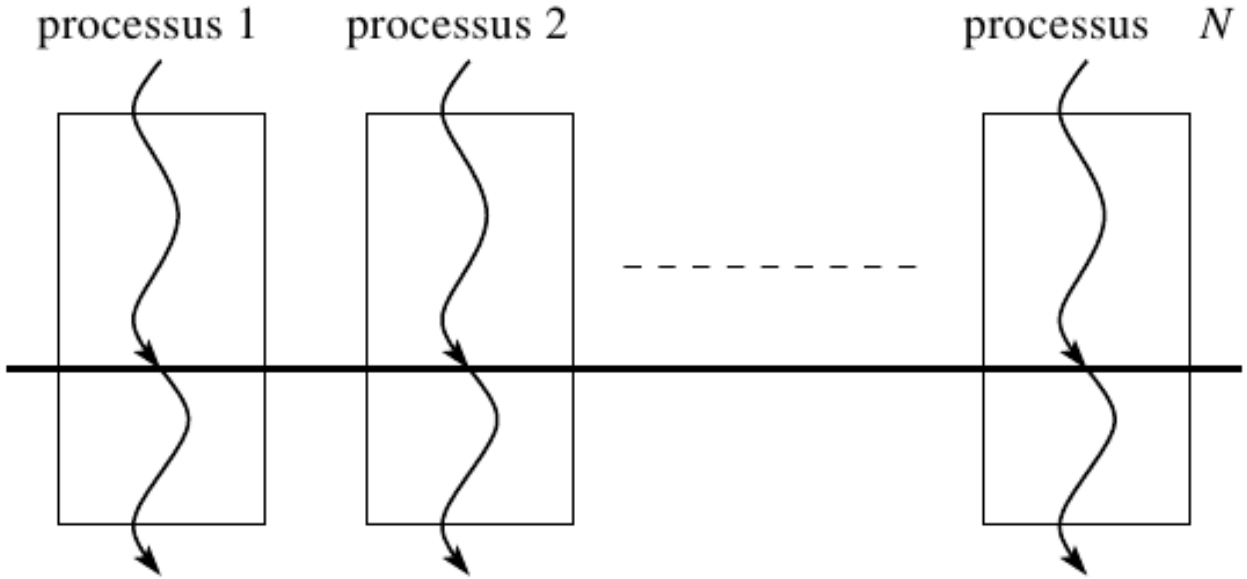
- La TâcheA obtient M1.
- La TâcheB obtient M2.
- La TâcheA attend pour obtenir M2 (qui est entre les mains de TâcheB).
- La TâcheB attend pour obtenir M1 (qui est entre les mains de TâcheA).

Dans cette situation, les deux tâches sont définitivement bloquées.

Prévention : Une méthode consiste à toujours acquérir les mutex dans le même ordre.

Sémaphore bloquant

- Un **sémaphore bloquant** est un **sémaphore qui est initialisé avec la valeur 0**.
- Ceci a pour effet de bloquer n'importe quel *thread* (ou processus) qui effectue `P(S)` tant qu'un autre *thread* (ou processus) n'aura pas fait un `V(S)`. Ce type d'utilisation est très utile lorsque l'on a besoin de **contrôler l'ordre d'exécution entre *threads* (ou processus)**.
- Cette utilisation des sémaphores permet de réaliser des **barrières de synchronisation**.



Bilan

Les sémaphores sont utilisés dans la programmation multi-tâche pour régler des problèmes :

- **d'accès à des ressources partagées entre tâches**
- **de synchronisation entre tâches**

Les sémaphores permettent notamment de résoudre les problèmes :

- des **producteurs/consommateurs**
- des **lecteurs/rédacteurs**

Les sémaphores ne protègent pas les programmeurs des **situations d'interblocage**.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Exemple de ressource : la file (1/2)

- Une **file (« *queue* » en anglais)** est une **structure de données basée sur le principe du « Premier entré, premier sorti », ou FIFO (*First In, First Out*)**, ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.
- Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.
- Une **file** est utilisée en général pour **mémoriser temporairement** (tampon ou *buffer*) **des données** (transactions, requêtes, mesures, ...) **qui doivent attendre pour être traitées**. Une file est systèmatiquement utilisée pour lire ou écrire sur des périphériques physiques (fichiers, *pool* d'impression, *socket* réseau, ...).

Exemple de ressource : la file (2/2)

Voici quelques fonctions communément utilisées pour manipuler des **files** :

- « Enfiler » : **ajoute** ou **dépose** un élément dans la file
- « Défiler » : renvoie le prochain élément de la file, et le **retire** de la file
- « La file est-elle vide ? » : renvoie « vrai » si la file est vide, « faux » sinon
- « La file est-elle pleine ? » : renvoie « vrai » si la file est pleine, « faux » sinon
- « Nombre d'éléments dans la file » : renvoie le nombre d'éléments présents dans la file
- « Taille de la file » : renvoie le nombre maximum d'éléments pouvant être déposés dans la file

Il existe aussi des files circulaires.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Exemple de ressource : la pile (1/2)

- Une **pile (« stack » en anglais)** est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (*Last In, First Out*)**, ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.
- Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.
- Une **pile** est utilisée en général pour **gérer un historique de données** (pages webs visitées, ...) ou **d'actions** (les fonctions « Annuler » de beaucoup de logiciels par exemple). La pile est utilisée aussi pour tous les paramètres d'appels et les variables locales des fonctions dans les langages compilés.

Exemple de ressource : la pile (2/2)

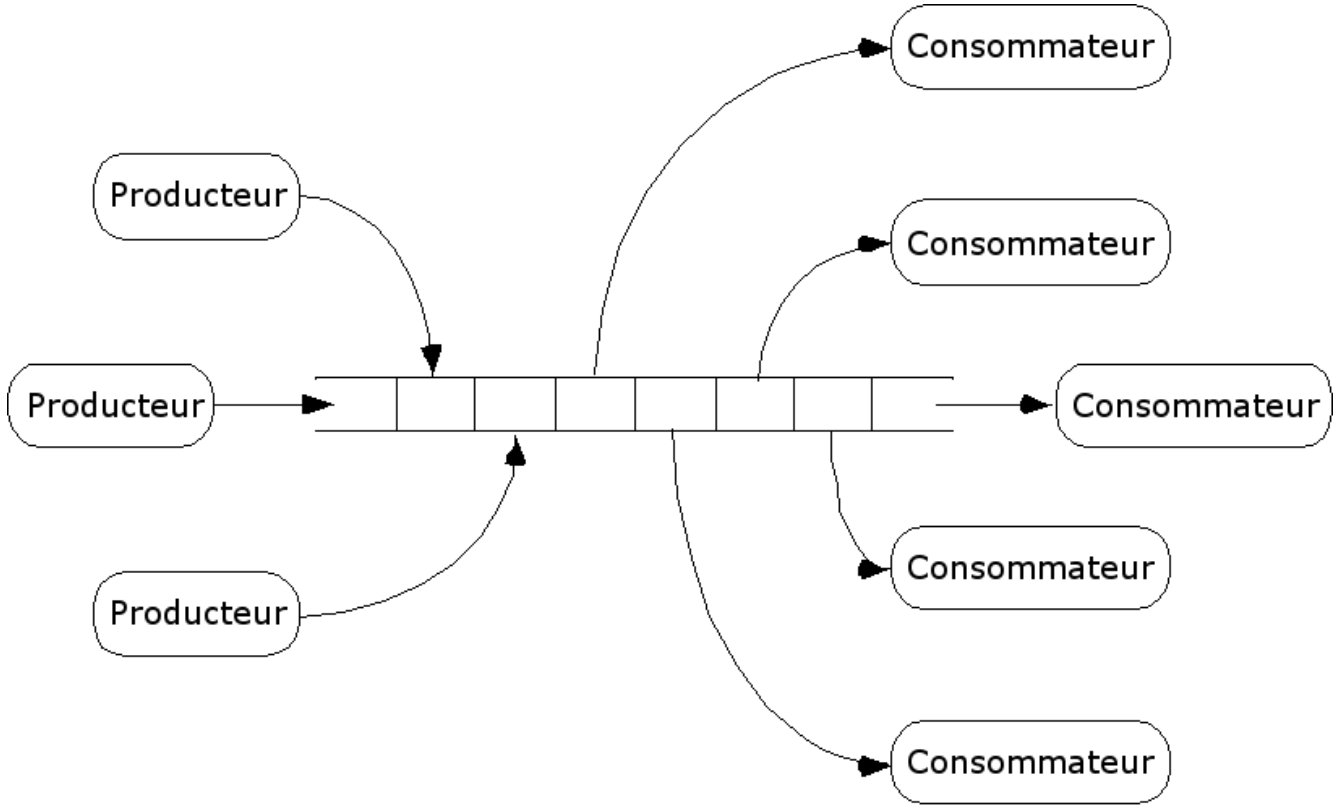
Voici quelques fonctions communément utilisées pour manipuler des **piles** :

- « Empiler » : **ajoute** ou **dépose** un élément sur la pile
- « Dépiler » : **enlève** un élément de la pile et le renvoie
- « La pile est-elle vide ? » : renvoie « vrai » si la pile est vide, « faux » sinon
- « La pile est-elle pleine ? » : renvoie « vrai » si la pile est pleine, « faux » sinon
- « Nombre d'éléments dans la pile » : renvoie le nombre d'éléments présents dans la pile
- « Taille de la pile » : renvoie le nombre maximum d'éléments pouvant être déposés dans la pile
- « Quel est l'élément de tête ? » : renvoie l'élément de tête sans le dépiler

Le problème des producteurs/consommateurs

Lorsque des processus légers (*threads*) souhaitent communiquer entre eux, ils peuvent le faire par l'intermédiaire d'une **file**. Il faut définir le comportement à avoir lorsqu' :

- un *thread* souhaite **lire** depuis la **file** lorsque celle-ci est vide et
- un *thread* souhaite **écrire** dans la **file** mais que celle-ci est pleine.



Le problème des lecteurs/rédacteurs

Un problème classique pouvant être résolu à l'aide des sémaphores est le problème des lecteurs/rédacteurs. Ce problème traite de **l'accès concurrent en lecture et en écriture** à une **ressource**.

- Plusieurs processus légers (*threads*) peuvent **lire** en même temps la **ressource**,
- mais il ne peut y avoir qu'un et un seul *thread* qui puisse **écrire** sur la **ressource**.