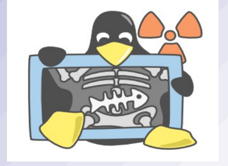
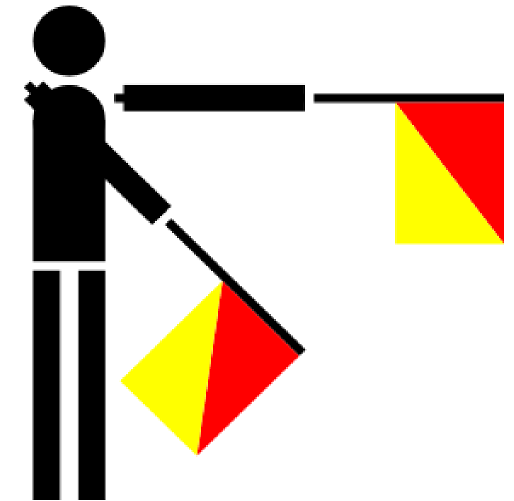
Programmation système : sémaphores



Un sémaphore permet de protéger une variable (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées dans un environnement de programmation concurrente.



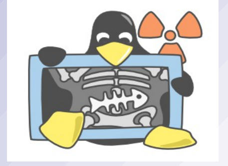
Le sémaphore a été inventé par Edsger Dijkstra.

Copyright 2011 tv <tvaira@free.fr>

Permission is granted to copy, distribute and/or modify this document under the terms of the **GNU Free Documentation License**, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover.

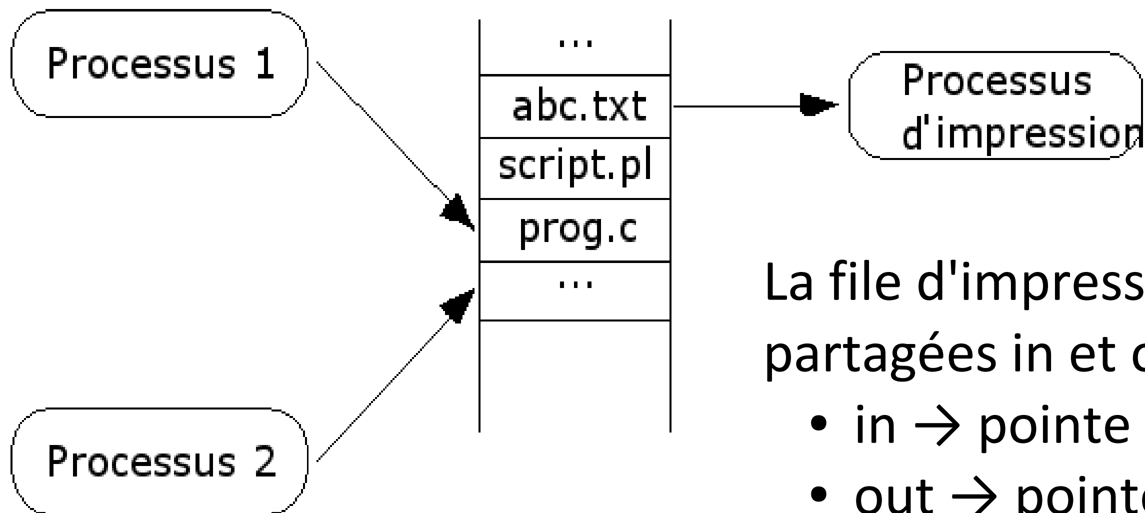
You can obtain a copy of the GNU General Public License :
write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Exemple introductif



- On considère le fonctionnement d'un *spool* d'impression.
- Problème : l'ordre d'exécution des instructions n'est pas garanti, donc par exemple l'impression du fichier n° 1 (F1) déposé par P1 ne se fera pas et l'état de la file est incohérent (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.

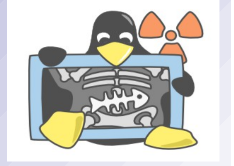
temps	P1	P2
(1)	file[in] := "F1";	
(2)		file[in] := "F2";
(3)		in++;
(4)	in++;	



La file d'impression est gérée à l'aide de 2 variables partagées in et out :

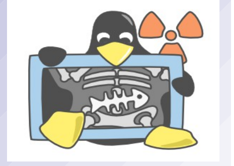
- in → pointe sur le premier emplacement libre
- out → pointe sur le prochain fichier à imprimer

Rappels : définitions



- **Section Critique :** C'est une partie de code telle que 2 processus ne peuvent s'y trouver au même instant.
- **Exclusion mutuelle :** Une ressource est en exclusion mutuelle si seul un processus peut utiliser la ressource à un instant donné.
- *Conditions de fonctionnement :* Plusieurs conditions sont nécessaires pour le bon fonctionnement de processus coopérants :
  - deux processus ne peuvent être, en même temps, en section critique,
  - aucune hypothèse n'est faite, ni sur la vitesse relative des processus, ni sur le nombre de processeurs,
  - aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus,
  - aucun processus ne doit attendre "trop longtemps" avant de pouvoir entrer en section critique.

Le masquage des interruptions

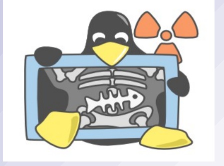


- L'idée consiste à empêcher la commutation de processus pendant l'exécution de la section critique. Ce résultat est atteint en masquant les interruptions qui provoquent le changement d'allocation du processeur. Le traitement des interruptions est alors différé :

```
Masquer_Interruption;
SectionCritique;
Restaurer_Interruption;
```

- Le masquage des interruptions est une technique utile dans le noyau, mais inappropriée pour les processus utilisateurs.

Les sémaphores

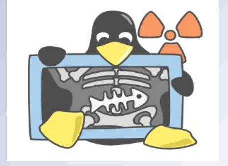


- Un sémaphore **S** est une **variable entière** qui n'est accessible qu'au travers de 3 opérations **Init, P et V**.

- La valeur d'un sémaphore est le **nombre d'unités de ressource** (exemple : imprimantes...) libres. S'il n'y a qu'une ressource, un sémaphore binaire avec les valeurs 0 ou 1 est utilisé.

- Les opérations 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.

Les opérations



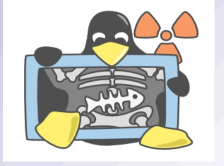
- **Init(S, valeur)** est seulement utilisé pour initialiser le sémaphore S avec une valeur. Cette opération ne doit être réalisée qu'une seule et unique fois.
- L'opération **P(S)** (du néerlandais *Proberen* signifiant tester et en français "Puis-je ?" ou éventuellement Prise ou Prendre) est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant.

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

- **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.

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

Sémaphore binaire

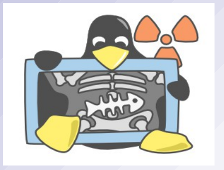


- Un sémaphore **binaire** est un sémaphore qui est initialisé avec la valeur 1.

- Ceci a pour effet de contrôler l'accès une ressource unique.

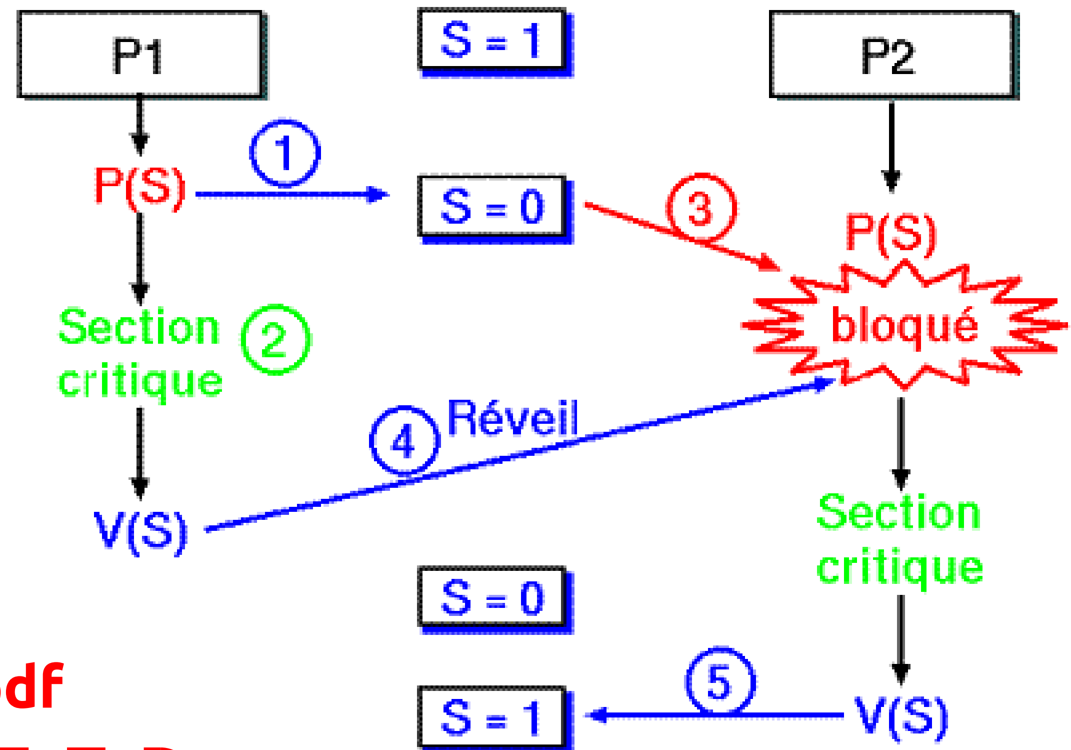
- Le sémaphore binaire permet l'**exclusion mutuelle** (mutex) : une ressource est en exclusion mutuelle si seul un processus peut utiliser la ressource à un instant donné.

Accès à une section critique



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

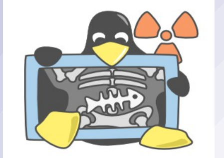
```
P(S);
SectionCritique;
V(S);
```



tp-sys-semaphores.pdf

Séquence 1 : Exemple

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
```

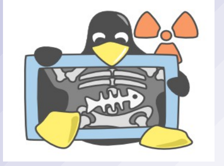
```
TâcheB :
  Obtenir M2
  Obtenir M1
  Action nécessitant les deux verrous
  Rendre M1
  Rendre M2
```

- 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).

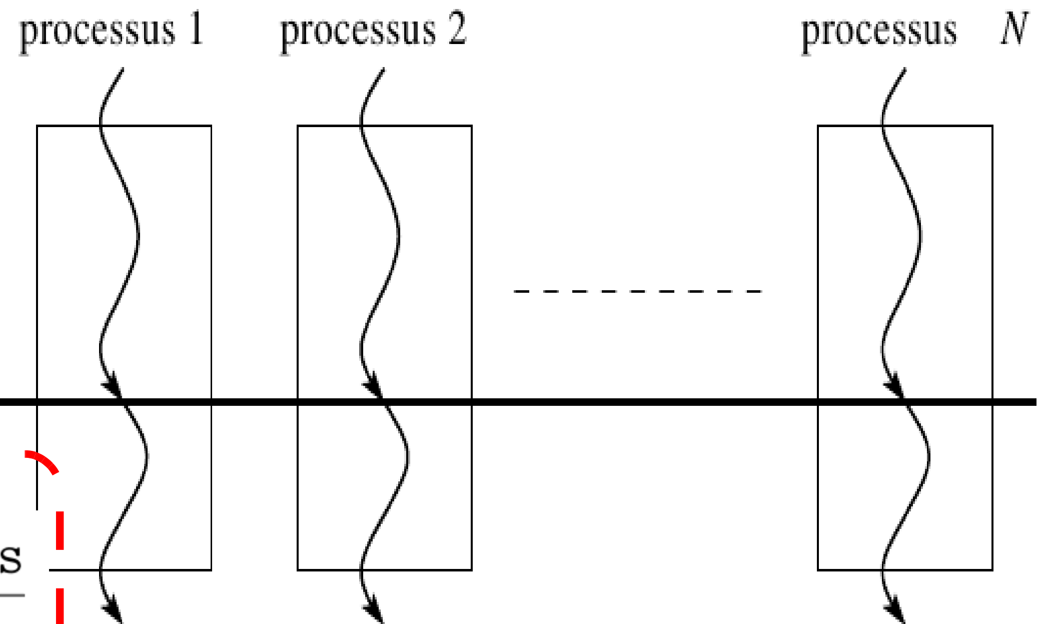
  Dans cette situation, les deux tâches sont définitivement bloquées.

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

Sémaphore bloquant



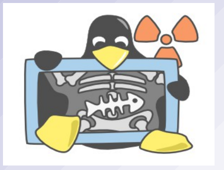
- Un sémaphore **bloquant** est un sémaphore qui est initialisé avec la valeur 0.
- Ceci a pour effet de bloquer n'importe quel thread qui effectue P(S) tant qu'un autre thread n'aura pas fait un V(S). Ce type d'utilisation est très utile lorsque l'on a besoin de contrôler l'ordre d'exécution entre threads.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



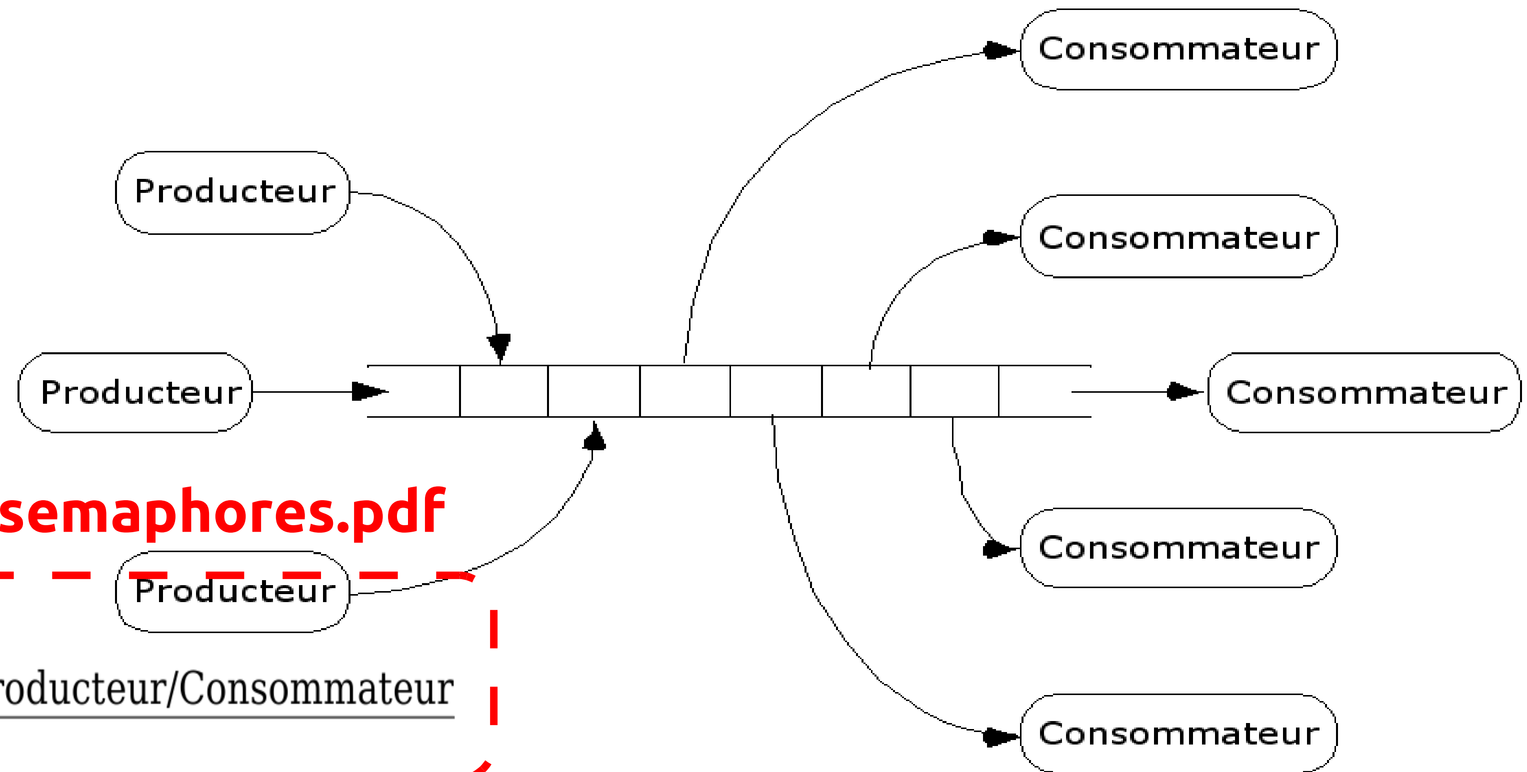
tp-sys-semaphores.pdf

Séquence 2 : Rendez-Vous

Producteur/Consommateur



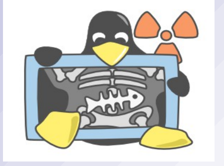
- Lorsque des processus légers souhaitent communiquer entre eux, ils peuvent le faire par l'intermédiaire d'une file. Il faut définir le comportement à avoir lorsqu'un thread souhaite lire depuis la file lorsque celle-ci est vide et lorsqu'un thread souhaite écrire dans la file mais que celle-ci est pleine.



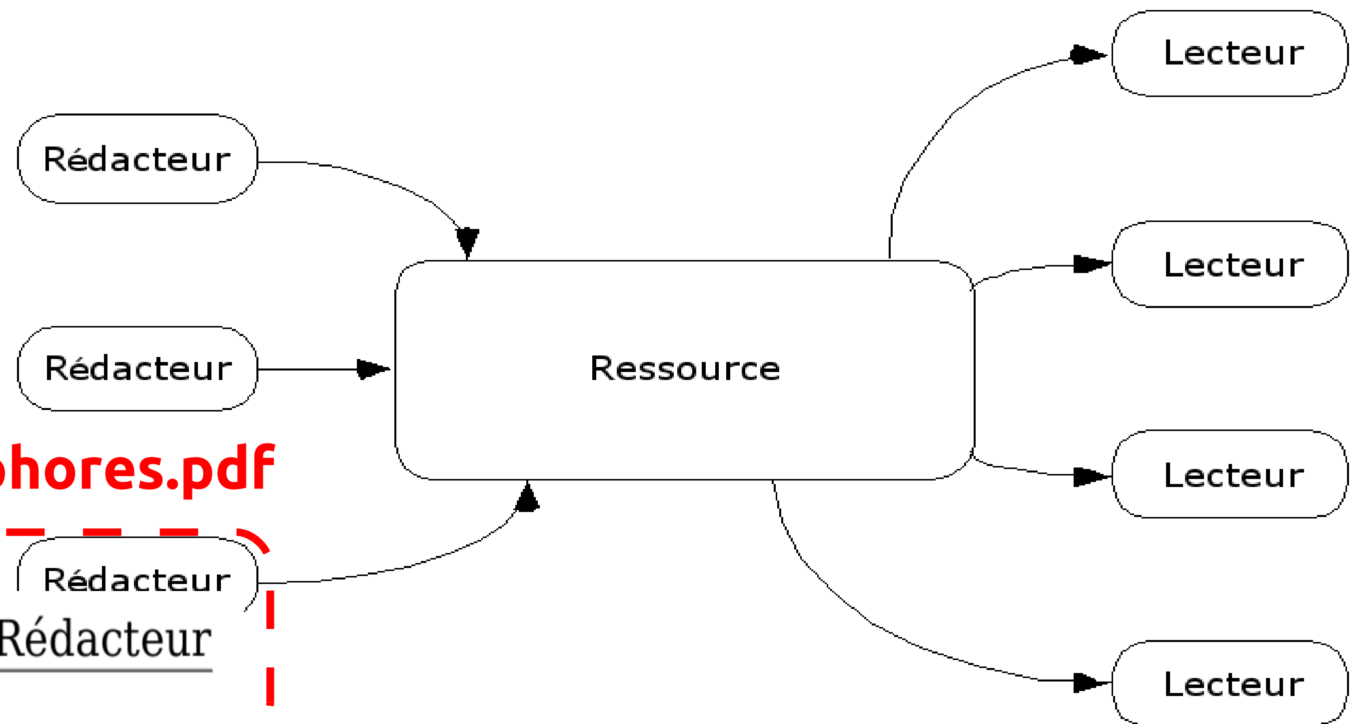
tp-sys-semaphores.pdf

Séquence 3 : Producteur/Consommateur

Lecteurs/Rédacteurs



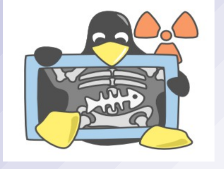
- Un problème classique pouvant être résolu à l'aide des sémaphores est le problème des lecteurs/rédacteurs.
- Ce problème traite de l'accès concurrent en lecture et en écriture à une ressource. Plusieurs processus légers (thread) peuvent lire en même temps la ressource, mais il ne peut y avoir qu'un et un seul thread en écriture.



tp-sys-semaphores.pdf

Séquence 4 : Lecteur-Rédacteur

Conclusion



- Il est conseillé avant de continuer de revoir :
  - Les principes et la terminologie associée aux sémaphores
  - Les exemples fournis pour les environnements Qt, Builder et Win32

- Il reste à voir entre autres : ordonnancement des processus, autres types de communications entre processus

tp-sys-semaphores.pdf

Bonus : Le problème du coiffeur endormi