Synchronisation entre processus

Dominique Bouillet et Michel Simatic

module CSC4508/M2

Avril 2018

Plan du document

1 Introduction

- Les problèmes de synchronisation sont légions dans la vie courante:
  - Quand on crédite son compte en banque, ce crédit ne doit pas être perdu parce qu'en parallèle la banque débite un chèque
  - Un parking de capacité N places ne doit pas laisser entrer $N + 1$ véhicules
  - Roméo et Juliette ne peuvent se prendre par la main que s'ils se retrouvent à leur rendez-vous
  - Robert et Raymonde font la vaisselle. Raymonde lave. Robert essuie. L'égouttoir les synchronise.
  - Certaines lignes de train possèdent des sections de voie unique. Sur ces sections, on ne peut avoir que des trains circulant dans un même sens à un instant donné
- On rencontre des problèmes similaires lorsqu'on utilise un système d'exploitation

1.1 Correspondance problèmes vie courante/informatique

Banque Problème d'*exclusion mutuelle* : une ressource ne doit être accessible que par une entité à un instant donné. Cas, par exemple, d'une zone mémoire contenant le solde d'un compte.

Parking Problème de *cohorte* : un groupe de taille bornée est autorisé à offrir/utiliser un service. Cas, par exemple, d'un serveur de connexions Internet qui ne doit pas autoriser plus de N connexions en parallèle.

Roméo et Juliette Problème de *passage de témoin* : on divise le travail entre des processus. Cas, par exemple, de 2 processus qui doivent s'échanger des informations à un moment donné de leur exécution avant de continuer.

Robert et Raymonde Problème de *producteurs/consommateurs* : un consommateur ne peut consommer que si tous les producteurs ont fait leur travail. Cas, par exemple, d'un processus chargé d'envoyer des tampons qui ont été remplis par d'autres processus.

Train Problème de *lecteurs/rédacteurs* où l'on doit gérer, de manière cohérente, une compétition entre différentes catégories d'entités. Cas, par exemple, d'une tâche de fond périodique de « nettoyage » qui ne peut se déclencher que quand les tâches principales sont inactives.

2 Sémaphore = Outil de base

2.1 Généralités

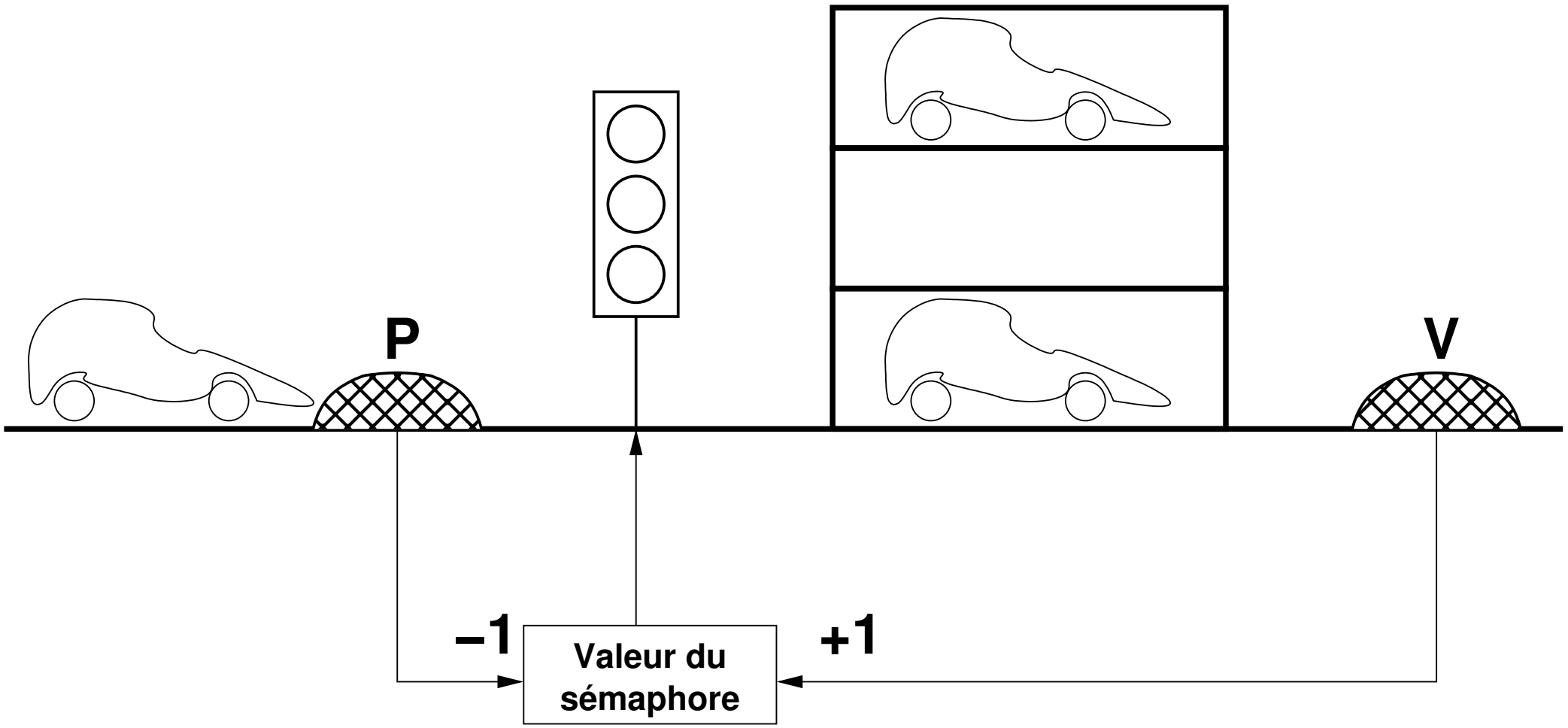
- Sémaphore = objet composé :
  - D'une variable (sa valeur)
  - D'une file d'attente (les processus bloqués)
- Primitives associées :
  - Initialisation (avec une valeur positive ou nulle)
  - Manipulation :
    - Prise (P ou Wait) = demande d'autorisation
    - Validation (V ou Signal) = fin d'utilisation
- Principe : sémaphore associé à une ressource
  - Prise = demande d'autorisation (Puis-je ?)

    si $valeur > 0$ accord, sinon blocage
  - Validation = restitution d'autorisation (Vas-y)

    si $valeur < 0$ déblocage d'un processus

2.2 Analogie

- Parking de N places contrôlé par un feu



2.3 Algorithmes P ET V

- Initialisation(sémaphore,n)

```
valeur[sémaphore] = n
```

- P(sémaphore)

```
valeur[sémaphore] = valeur[sémaphore] - 1
si (valeur[sémaphore] < 0) alors
   étatProcessus = Bloqué
   mettre processus en file d'attente
finSi
invoquer l'ordonnanceur
```

- V(sémaphore)

```
valeur[sémaphore] = valeur[sémaphore] + 1
si (valeur[sémaphore] <= 0) alors
   extraire processus de file d'attente
   étatProcessus = Prêt
finSi
invoquer l'ordonnanceur
```

3 Résolution de problèmes de synchronisation typiques

3.1 Exclusion mutuelle

- Permet la gestion d'accès concurrent à des ressources partagées
- Principe :
  - Sémaphore `mutex` initialisé à 1
  - Primitive P en début de section critique
  - Primitive V en fin de section critique
- Exemple :
  - Sémaphore `mutex` initialisé à 1
  -

```
   Prog1                Prog2
P(mutex)             P(mutex)
 x=lire (cpte)        x=lire (cpte)
 x = x + 10           x = x - 100
 écrire (cpte=x)      écrire (cpte=x)
V(mutex)             V(mutex)
```

3.2 Cohorte

- Permet la coopération d'un groupe de taille maximum donnée
- Principe :
  - Sémaphore `parking` initialisé à N
  - Primitive P en début de besoin
  - Primitive V en fin de besoin
- Exemple :
  - Sémaphore `parking` initialisé à N
  - `Prog vehicule`

```
...
P(parking)
|...
V(parking)
...
```

3.3 Passage de témoin

- Permet la coopération par division du travail entre processus
- 3 types :
  - Envoi de signal
  - Rendez-vous entre 2 processus
  - Appel procédural entre processus

3.3.1 Envoi de signal

- Permet à un processus d'envoyer un « top » à un autre pour signaler la présence/disponibilité d'une information
- Principe :
  - Sémaphore `top` initialisé à 0
  - Primitive P pour attente « top »
  - Primitive V pour signal « top »
- Exemple :
  - Sémaphore `top` initialisé à 0
  -

```
   Prog1                Prog2
...                     ...
calcul(info)            P(top)
V(top)                  utilisation(info)
...                     ...
```

3.3.2 Rendez-vous entre deux processus

- Permet à deux processus d'établir un point de synchronisation
- Principe :
  - Sémaphore `romeo` initialisé à 0
  - Sémaphore `juliette` initialisé à 0
  -

```
   Prog1                Prog2
...                  ...
V(juliette)          V(romeo)
P(romeo)             P(juliette)
...                  ...
```

3.3.3 Appel procédural entre processus

- Permet à un processus de faire un « appel de procédure », alors que le code de cette procédure est localisé dans un autre processus
- Principe :
  - Sémaphore `appel` initialisé à 0
  - Sémaphore `retour` initialisé à 0
  -

```
    Serveur                          Client
répéter                          ...
  P(appel)                       preparationParamAppel()
  analyseParamAppel()            V(appel)
  ...                            P(retour)
  préparationParamRetour()       analyseParamRetour()
  V(retour)                      ...
finRépéter
```

3.4 Producteurs/Consommateurs

3.4.1 Objectif

- Permettre le contrôle de flux entre un (ou des) producteur(s) et un (ou des) consommateur(s) dans le cas où ils communiquent via un tampon mémoire de N cases

- 1. Exécution `Produc` : il produit `info0`

info0				

2. Exécution `Produc` : il produit `info1`

info0	info1			

3. Exécution `Conso` : il consomme `info0`

	info1			

4. Exécution `Produc` : il produit `info2`

	info1	info2		

3.4.2 Principe

- Sémaphore `infoPrete` initialisé à 0
- Sémaphore `placeDispo` initialisé à N
-

```
   Produc                    Conso
répéter                   répéter
  ...                       P(infoPrete)
  calcul(info)              extraire(info)
  P(placeDispo)             V(placeDispo)
  déposer(info)             utiliser(info)
  V(infoPrete)            finRépéter
  ...
finRépéter
```

3.4.3 Déposer et extraire

- Le tampon ne peut s'étendre à l'infini → Tampon géré de façon circulaire, c'est-à-dire que quand dernière case utilisée, retour à la première
- Il faut :
  - un indice de dépot `iDépot`
  - un indice d'extraction `iExtrait`
  - ajouter à l'initialisation `iDépot = 0` et `iExtrait = 0`
- `déposer(info)`

```
tampon[iDépot] = info
iDépot = (iDépot + 1) modulo N
```

- `extraire(info)`

```
info = tampon[iExtrait]
iExtrait = (iExtrait + 1) modulo N
```

3.4.4 Exemple

1. On suppose qu'à un instant donné, le tampon est dans l'état suivant :
   - Tampon :

			info8	

   - $valeurSem[placeDispo] = 4$ et $valeurSem[infoPrete] = 1$
   - $iDépot = 4$ et $iExtrait = 3$
2. Exécution `Produc` : il produit `info9`
   - Tampon :

			info8	info9

   - $valeurSem[placeDispo] = 3$ et $valeurSem[infoPrete] = 2$
   - $iDépot = 0$ et $iExtrait = 3$
3. Exécution `Produc` : il produit `infoA`
   - Tampon :

infoA			info8	info9

   - $valeurSem[placeDispo] = 2$ et $valeurSem[infoPrete] = 3$
   - $iDépot = 1$ et $iExtrait = 3$
4. Exécution `Conso` : il consomme `info8`

- Tampon :

- $valeurSem[placeDispo] = 3$ et $valeurSem[infoPrete] = 2$
- $iDépot = 1$ et $iExtrait = 4$

3.4.5 K producteurs

- Ce modèle semble rester valable pour plusieurs producteurs (ou plusieurs consommateurs) en termes de synchronisation...
- ...mais en fait risque d'information perdue à cause des indices iDépot (pour K producteurs) ou iExtrait (pour P consommateurs) qui sont des variables globales (*cf.* exemple)

3.4.6 Exemple de problème avec 2 producteurs

- soit un tampon avec $N = 5$ et deux dépots effectués sans extraction
- L'état courant est alors :
  - Tampon :

info0	info1			

  - $valeurSem[placeDispo] = 3$ et $valeurSem[infoPrete] = 2$
  - $iDépot = 2$ et $iExtrait = 0$

Exemple de problème avec 2 producteurs (suite)

1. Début d'exécution de `Produc1` :
   `P(placeDispo);tampon[iDépot] = infoP1;...`
   - Tampon :

info0	info1	infoP1		

   - $valeurSem[placeDispo] = 2$ et $valeurSem[infoPrete] = 2$
   - $iDépot = 2$ et $iExtrait = 0$

2. Exécution complète de `Produc2` : `P(placeDispo);tampon[iDépot] = infoP2;iDépot=(iDépot+1) modulo N;V(infoPrete)`
   - Tampon :

info0	info1	infoP2		

   - $valeurSem[placeDispo] = 1$ et $valeurSem[infoPrete] = 3$
   - $iDépot = 3$ et $iExtrait = 0$

3. Fin d'exécution de `Produc1` :
   `...;iDépot=(iDépot+1) modulo N;V(infoPrete)`
   - Tampon :

info0	info1	infoP2	i!*?#	

- $valeurSem[placeDispo] = 1$ et $valeurSem[infoPrete] = 4$
- $iDépot = 4$ et $iExtrait = 0$

3.4.7 Solution complète

- Il faut une exclusion mutuelle sur `déposer()` et `extraire()` autour de `iDépot` et `iExtrait`
- Pour K producteurs :
  - Sémaphore `mutex` initialisé à 1
  - déposer(info)

```
P(mutex)
|tampon[iDépot] = info
|iDépot = (iDépot + 1) modulo N
V(mutex)
```

- Pour P consommateurs, idem pour `extraire`
- Pour K producteurs et P consommateurs, utiliser deux sémaphores `mutexP` et `mutexC`

3.5 Lecteurs/rédacteurs

3.5.1 Objectif

- Permettre une compétition cohérente entre deux types de processus (les « lecteurs » et les « rédacteurs ») :
  - Plusieurs lecteurs peuvent accéder simultanément à la ressource
  - Les rédacteurs sont exclusifs entre eux pour leur exploitation de la ressource
  - Un rédacteur est exclusif avec les lecteurs

3.5.2 Solution de base

- Sémaphore `mutexG` initialisé à 1
- Sémaphore `mutexL` initialisé à 1
- Entier `NL` initialisé à 0
-

```
   Lecteur                    Rédacteur
P(mutexL)                  P(mutexG)
|NL = NL + 1                | ecrituresEtLectures()
|si NL == 1 alors          V(mutexG)
|   P(mutexG)
|finSi
V(mutexL)
lectures()
P(mutexL)
|NL = NL - 1
|si NL == 0 alors
|   V(mutexG)
|finSi
V(mutexL)
```

3.5.3 Analyse

- La solution de base fonctionne, mais on constate qu'il y a une possibilité de famine pour les rédacteurs.
- Pour éviter cette famine, on ajoute les contraintes suivantes :
  - Si la ressource est utilisée par un lecteur :
    - Tout écrivain est mis en attente.
    - Tout lecteur est accepté s'il n'y a pas d'écrivain en attente.
    - Tout lecteur est mis en attente s'il y a un écrivain en attente.

3.5.4 Solution avec priorités égales

- Sémaphore `mutexG` initialisé à 1
- Sémaphore `mutexL` initialisé à 1
- Sémaphore `fifo` initialisé à 1
- Entier `NL` initialisé à 0
-

```
   Lecteur                  Rédacteur
P(fifo)                  P(fifo)
 P(mutexL)                 P(mutexG)
  NL = NL + 1            V(fifo)
  si NL == 1 alors       ecrituresEtLectures()
     P(mutexG)           V(mutexG)
  finSi
 V(mutexL)
V(fifo)
lectures()
P(mutexL)
 NL = NL - 1
 si NL == 0 alors
    V(mutexG)
 finSi
V(mutexL)
```

4 Interblocage

4.1 Introduction

- Interblocage (Deadlock) = toute situation telle que deux processus au moins sont chacun en attente d'une ressource non partageable déjà allouée à l'autre
- Exemple : exclusion mutuelle sur deux ressources différentes
  - Sémaphore `mutex1` initialisé à 1
  - Sémaphore `mutex2` initialisé à 1
  -

```
   Prog1                          Prog2
...                            ...
P(mutex1)                      P(mutex2)
|accès à ressource 1           |accès à ressource 2
|P(mutex2)                     |P(mutex1)
||accès à ressource 2          ||accès à ressource 1
|V(mutex2)                     |V(mutex1)
V(mutex1)                      V(mutex2)
...                            ...
```

  - Les deux programmes se bloqueront mutuellement si `Prog1` fait `P(mutex1)` alors que simultanément `Prog2` fait `P(mutex2)`

4.2 Généralités

- Synonymes d'interblocage : verrou mortel ou étreinte fatale
- Conditions nécessaires et suffisantes (Coffman, 1971)

  4 CNS simultanées :

  1. Ressources en exclusion mutuelle (ressources concernées non partageables)
  2. Processus en attente (processus conserve les ressources allouées)
  3. Non préemption des ressources (ressources concernées non préemptibles)
  4. Chaîne circulaire de processus bloqués (généralisation des demandes réciproques)
- 3 stratégies de lutte contre les blocages :
  - Prévention (statique)
  - Détection avec guérison
  - Évitement (prévention dynamique)

5 Mise en oeuvre dans un système d'exploitation

- Sémaphores (*cf.* chapitres « Communications inter-processus » et « Threads »)
  - Sémaphores IPC
  - Sémaphores Posix
- Moniteurs (ou conditions ou variables-condition)
  - Le moniteur ajoute le type `condition` et une file d'attente associée à chaque variable de type `condition` ainsi que des primitives `wait x`, `signal x` (`x` étant une condition).
  - Si besoin, un processus se bloque par `wait` et libère le moniteur
  - Un processus réveille un des processus en attente par `signal`
  - Il peut également réveiller tous les processus en attente par `broadcast`

Bibliographie du chapitre