

## Troisième partie

# Sémaphore



## Contenu de cette partie

- Présentation d'un objet de synchronisation élémentaire (sémaphore)
- Patrons de conception élémentaires utilisant les sémaphores
- Exemple récapitulatif (schéma producteurs/consommateurs)
- Schémas d'utilisation pour le contrôle fin de l'accès aux ressources partagées
- Mise en œuvre des sémaphores



# Plan

- 1 Spécification
  - Définition
  - Spécification intuitive
  - Spécification formelle : Hoare
  - Ordonnancement
- 2 Applications
  - Méthodologie
  - L'exclusion mutuelle
  - L'allocation de ressources critiques
  - Producteurs/consommateurs
- 3 Implantation
  - Sémaphore général à partir de verrous
  - Système d'exploitation
  - L'inversion de priorité



# But

- Fournir un moyen *simple*, élémentaire, de contrôler les effets des interactions entre activités
  - isoler (modularité) : atomicité
  - spécifier des interactions précises : synchronisation
- Exprimer ce contrôle par des interactions sur un *objet partagé* (indépendant des activités en concurrence) plutôt que par des interactions entre activités (dont le code et le comportement seraient alors interdépendants)



## Définition – Dijkstra 1968

Principes directeurs :

- Abstraction de la file d'attente des activités bloquées
- Manipuler des objets de synchronisation, pas des activités

### Définition

Un sémaphore encapsule un entier, avec une contrainte de **positivité**, et deux opérations **atomiques** d'incrémentement et de décrémentement.

- Contrainte de positivité : l'entier est toujours positif ou nul.
- opération **down** : décrémente le compteur s'il est strictement positif ; bloque s'il est nul en attendant de pouvoir le décrémenter.
- opération **up** : incrémente le compteur.

# Spécification intuitive

## Définition

Un sémaphore est un tas de cailloux avec deux opérations :

- Prendre un caillou, en attendant si nécessaire qu'il y en ait ;
- Déposer un caillou.

Attention :

- les cailloux sont anonymes et illimités : une activité peut déposer un caillou sans en avoir pris ;
- il n'y a pas de lien entre le caillou déposé et l'activité déposante ;
- lorsqu'une activité dépose un caillou et qu'il existe des activités en attente, **une seule** d'entre elles peut prendre un caillou.



# Spécification formelle : Hoare

## Définition

Un sémaphore  $S$  encapsule un entier  $cnt$  tel que

$$\begin{array}{ccc} \text{init} & \Rightarrow & S.cnt \geq 0 \\ \{S.cnt = k \wedge k > 0\} & S.down() & \{S.cnt = k - 1\} \\ \{S.cnt = k\} & S.up() & \{S.cnt = k + 1\} \end{array}$$

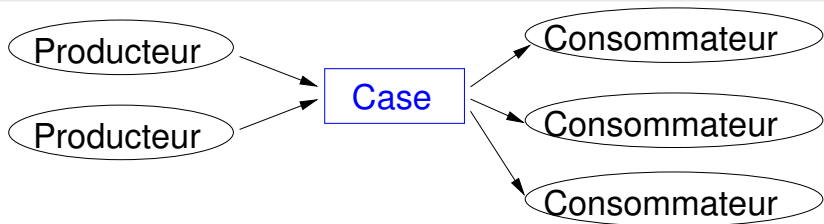
# Propriétés

- Si la précondition de  $S.down()$  n'est pas vérifiée, l'activité est retardée ou bloquée.
- Quand une activité, via l'opération  $up$ , rend vraie la précondition de  $S.down()$  et qu'il existe au moins une activité bloquée sur  $down$ , **une** telle activité est débloquée (et décrémente le compteur).
- **invariant**  $S.cnt = S.cnt_{init} + \#up - \#down$   
où  $\#up$  et  $\#down$  sont le nombre d'opérations  $up$  et  $down$  effectuées.





## Producteurs/consommateurs simplifié



- échange des données via une **unique** case (zone mémoire partagée)
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs
- objectifs : ne pas écraser une case occupée, une unique lecture consommatrice par case, attendre pour déposer si plein, attendre pour retirer si vide



# Producteurs/consommateurs

```
case := nil // tampon partagé  
Sémaphores : vide := 1, plein := 0
```

déposer(item)

```
vide.down()  
{ pré : case libre }  
case ← item  
{ post : case occupée }  
plein.up()
```

retirer(\*item)

```
plein.down()  
{ pré : case occupée }  
*item ← case  
{ post : case libre }  
vide.up()
```

Autres noms des opérations :

P	Probeer (essayer [de décrémente])	down	wait/attendre	acquire/prendre
V	Verhoog (incrémenter)	up	signal(er)	release/libérer



# Ordonnement

Plusieurs stratégies de déblocage :

- First-In-First-Out = par ordre chronologique d'arrivée
- Priorité des activités demandeuses
- Indéfinie

Les implantations courantes sont en général sans stratégie définie : avec une primitive rapide mais non équitable, on peut implanter (laborieusement) une solution équitable, mais avec une primitive lente et équitable, on **ne peut pas** implanter une solution rapide (et inéquitable).



## Variante : `down` non bloquant

### opération `tryDown`

$$\left\{ S.cnt = k \right\} r \leftarrow S.tryDown() \left\{ \begin{array}{l} (k > 0 \wedge S.cnt = k - 1 \wedge r) \\ \vee (k = 0 \wedge S.cnt = k \wedge \neg r) \end{array} \right\}$$

Conduit à de l'attente active.  
Généralement utilisé à tort.

# Plan

- 1 Spécification
  - Définition
  - Spécification intuitive
  - Spécification formelle : Hoare
  - Ordonnancement
- 2 Applications
  - Méthodologie
  - L'exclusion mutuelle
  - L'allocateur de ressources critiques
  - Producteurs/consommateurs
- 3 Implantation
  - Sémaphore général à partir de verrous
  - Système d'exploitation
  - L'inversion de priorité



## Schémas standards

### Contrôle du degré de parallélisme

sémaphore accès := N

accès.down() zone(s) contrôlée(s) accès.up()

### Événements

Attendre/signaler un événement :

- associer un sémaphore à l'événement : occurrenceE (généralement initialisé à 0)
- signaler la présence de l'événement : occurrenceE.up()
- attendre et consommer une occurrence de l'événement : occurrenceE.down()

## Schémas moins standards

### Rendez-vous entre deux activités

```
sémaphore aArrivé := 0
```

```
sémaphore bArrivé := 0
```

*Activité A*

⋮

```
aArrivé.up()
```

```
bArrivé.down()
```

⋮

*Activité B*

⋮

```
bArrivé.up()
```

```
aArrivé.down()
```

⋮



## Schémas moins standards

### Rendez-vous à N activités (barrière)

Pour passer la barrière, une activité doit attendre que les  $N - 1$  autres activités l'aient atteinte.

```
barrière = array 0..N-1 of Semaphore := {0,...};
```

```
-- Protocole de passage pour l'activité k  
for i := 0..N-1 do barrière[k].up(); end;  
for i := 0..N-1 do barrière[i].down(); end;
```



## Pré/post-conditions

### Précondition

Précondition de l'action qui suit =

- état qui doit être vrai pour pouvoir faire l'action,
- ou événement qui doit être survenu pour pouvoir faire l'action.

sém.down

### Postcondition

Postcondition de l'action précédente

- état garanti après terminaison de l'action,
- ou événement qui survient après l'action.

sém.up

# L'exclusion mutuelle

## Algorithme

```
mutex : global semaphore := 1;
```

```
mutex.down()
```

```
    section critique
```

```
mutex.up()
```

## Sémaphore booléen – Verrou

### Définition

Sémaphore  $S$  encapsulant un booléen  $b$  tel que

$$\begin{array}{lll} \{S.b\} & S.down() & \{\neg S.b\} \\ \{true\} & S.up() & \{S.b\} \end{array}$$

- Un sémaphore booléen est différent d'un sémaphore entier initialisé à 1 : plusieurs *up* consécutifs sont équivalents à un seul.
- Souvent nommé **verrou/lock**
- Opérations *down/up* = *lock/unlock* ou *acquiere/release*



## Verrou lecture/écriture

Une ressource peut être utilisée :

- concurremment par plusieurs lecteurs (plusieurs lecteurs simultanément) ;
- exclusivement par un rédacteur (pas d'autre rédacteur, pas d'autre lecteur).

Souvent rencontré sous la forme de **verrou lecture/écriture** (read-write lock).

Permet l'isolation des modifications avec un meilleur parallélisme que l'exclusion mutuelle.



## Allocateur (simplifié) de ressources

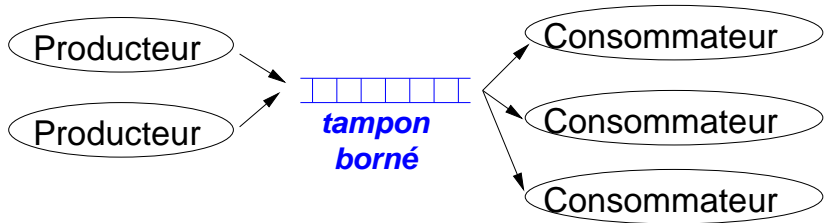
### Définition

- $N$  ressources critiques, équivalentes, réutilisables
  - usage exclusif des ressources
  - opération allouer **une** ressource
  - opération libérer une ressource précédemment obtenue
  - bon comportement d'une activité : pas deux demandes d'allocation consécutives sans libération intermédiaire
- ⇒ trivialement implanté par un sémaphore avec  $N$  jetons

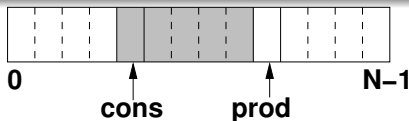
Attention : le problème où allouer demande plusieurs ressources est plus dur ! Idem, si plusieurs allocations consécutives.



## Producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs
- objectifs : ne pas écraser une case occupée, une unique lecture consommatrice par case, attendre pour déposer si plein, attendre pour retirer si vide



déposer(l)

```
vide.down()
{ pré : ∃ places libres }
  mutex.down()
  { pré : section critique }
    tampon[prod] ← 1
    prod ← prod + 1 mod N
  { post : fin SC }
  mutex.up()
{ post : ∃ places occupées }
plein.up()
```

retirer(\*l)

```
plein.down()
{ pré : ∃ places occupées }
  mutex.down()
  { pré : section critique }
    *l ← tampon[cons]
    cons ← cons + 1 mod N
  { post : fin SC }
  mutex.up()
{ post : ∃ places libres }
vide.up()
```

Sémaphores : mutex := 1, vide := N, plein := 0





# Plan

- 1 Spécification
  - Définition
  - Spécification intuitive
  - Spécification formelle : Hoare
  - Ordonnancement
- 2 Applications
  - Méthodologie
  - L'exclusion mutuelle
  - L'allocateur de ressources critiques
  - Producteurs/consommateurs
- 3 Implantation
  - Sémaphore général à partir de verrous
  - Système d'exploitation
  - L'inversion de priorité



# Sémaphore général à partir de verrous

## Algorithme

```
Sg = ⟨cnt := ?,  
      mutex := BinarySemaphore(true),  
      accès := BinarySemaphore(cnt > 0)⟩ // verrous  
  
Sg.down() =  Sg.accès.lock  
             Sg.mutex.lock  
             S.cnt ← S.cnt - 1  
             si S.cnt ≥ 1 alors Sg.accès.unlock  
             Sg.mutex.unlock  
  
Sg.up() =    Sg.mutex.lock  
            S.cnt ← S.cnt + 1  
            si S.cnt = 1 alors Sg.accès.unlock  
            Sg.mutex.unlock
```

→ les sémaphores binaires ont (au moins) la même puissance d'expression que les sémaphores généraux



# Implantation d'un sémaphore

Gestion des activités fournissant :

- l'exclusion mutuelle (cf partie II)
- le blocage (suspension) et déblocage (reprise) des activités

## Implantation

```
Sémaphore =  $\langle$  int nbjetons ;  
File<Activité> bloquées  $\rangle$ 
```



## Algorithme

```
S.down() =  entrer en excl. mutuelle
            si S.nbjets = 0 alors
                insérer self dans S.bloquées
                suspendre l'activité courante
            sinon
                S.nbjets ← S.nbjets - 1
            finsi
            sortir d'excl. mutuelle
```

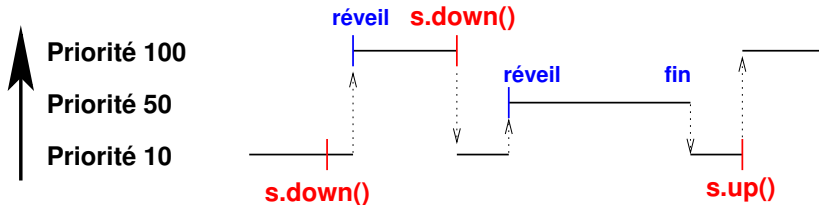
---

```
S.up() =   entrer en excl. mutuelle
            si S.bloquées ≠ vide alors
                actRéveillée ← extraire de S.bloquées
                débloquer actRéveillée
            sinon
                S.nbjets ← S.nbjets + 1
            finsi
            sortir d'excl. mutuelle
```

## Sémaphores et priorités

Temps-réel  $\Rightarrow$  priorité  $\Rightarrow$  sémaphore non-FIFO.

**Inversion de priorités** : une activité moins prioritaire bloque indirectement une activité plus prioritaire.



## Solution à l'inversion de priorité

- Plafonnement de priorité (priority ceiling) : monter **systematiquement** la priorité d'une activité verrouilleuse à la priorité maximale des activités **potentiellement** utilisatrices de cette ressource.
  - Nécessite de connaître a priori les demandeurs
  - Augmente la priorité même en absence de conflit
  - + Simple et facile à implanter
  - + Prédicible : la priorité est associée au sémaphore (= à la ressource)
- Héritage de priorité : monter **dynamiquement** la priorité d'une activité verrouilleuse à celle du demandeur.
  - + Limite les cas d'augmentation de priorité aux cas de conflit
  - Nécessite de connaître les possesseurs d'un sémaphore
  - Dynamique  $\Rightarrow$  comportement moins prédictible



# Conclusion

Le concept de sémaphore est

- + simple
- + facile à comprendre
- + performant
- délicat à l'usage
  - schémas génériques

