Troisième partie

Sémaphores

Contenu de cette partie

- présentation d'un objet de synchronisation « minimal » (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

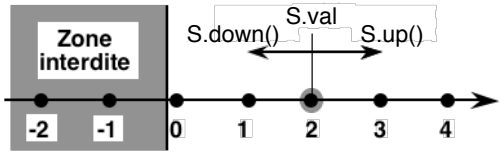
But

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

Définition – Dijkstra 1968

Un sémaphore S est un objet dont

- l'état val est un attribut entier *privé* (l'état est encapsulé)
- l'ensemble des états permis est contraint par un invariant (*contrainte de synchronisation*) :
  invariant $S.val \geq 0$ (l'état doit toujours rester positif ou nul)
- l'interface fournit deux opérations principales :
  - *down* : bloque si l'état est nul, décrémente l'état s'il est > 0
  - *up* : incrémente l'état
    $\rightarrow$ permet de débloquer un éventuel processus bloqué sur `down`
  - les opérations `down` et `up` sont *atomiques*



- *Autre opération* : constructeur (et/ou initialisation)
  $S = new\ Semaphore(v_0)$ (ou $S.init(v_0)$)
  (crée et) initialise l'état de S à v_0
- *Autres noms des opérations*

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

Modèle intuitif

Un sémaphore peut être vu comme un tas de jetons avec 2 actions

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

Attention

- les jetons sont anonymes et illimités : un processus peut déposer un jeton sans en avoir pris ;
- il n'y a pas de lien entre le jeton déposé et le processus déposeur ;
- lorsqu'un processus dépose un jeton et que des processus sont en attente, *un seul* d'entre eux peut prendre ce jeton.

Définition formelle : Hoare

Définition

Un sémaphore S encapsule un entier val tel que

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

Remarques

- Si la précondition de $S.down()$ est fausse, le processus attend.
- Si l'exécution de l'opération *up*, rend vraie la précondition de $S.down()$ et qu'il y a au moins une activité bloquée sur *down*, une telle activité est débloquée (et décrémente le compteur).
- l'invariant du sémaphore peut aussi s'exprimer à partir des nombres $\#down$ et $\#up$ d'opérations *down* et *up* effectuées :
  invariant $S.val = S.val_{\text{init}} + \#up - \#down$

Remarques

1. Lors de l'exécution d'une opération *up*, s'il existe plusieurs processus en attente, la politique de choix du processus à débloquer peut être :
   - par ordre chronologique d'arrivée (FIFO) : équitable
   - associée à une priorité affectée aux processus en attente
   - indéfinie.
     C'est le cas le plus courant : 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.
2. Variante : *down* non bloquant (*tryDown*)

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

Attention aux mauvais usages : incite à l'attente active.

Sémaphore binaire (booléen) – Verrou

Définition

Sémaphore S encapsulant un entier b tel que

$$\begin{array}{ccc} \{S.b = 1\} & S.down() & \{S.b = 0\} \\ \{true\} & S.up() & \{S.b = 1\} \end{array}$$

- Un sémaphore binaire est différent d'un sémaphore entier initialisé à 1.
- Souvent nommé verrou/lock
- Opérations down/up = lock/unlock ou acquire/release

Plan

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(--); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

     section critique
```

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(1); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

mutex.down()
     section critique
mutex.up()
```

Schémas d'utilisation essentiels (1/4)

Généralisation : contrôle du degré de parallélisme

Algorithme

Pour limiter à *Max* le nombre d'accès simultanés à la ressource *R* :

- Objet partagé :

```
global accèsR = new Semaphore(Max)
```

- Protocole d'accès à la ressource *R* (pour *chaque* processus) :

```
accèsR.down()
     accès à la ressource R
accèsR.up()
```

Règle de conception

- Identifier les portions de code où le parallélisme doit être limité
- Définir un sémaphore pour contrôler le degré de parallélisme
- Encadrer ces portions de code par down/up sur ce sémaphore

Schémas d'utilisation essentiels (2/4)

Synchronisation élémentaire : attendre/signaler un événement *E*

- Objet partagé :

```
occurrenceE = new Semaphore(0) // initialisé à 0
```

- attendre une occurrence de *E* : `occurrenceE.down()`
- signaler l'occurence de l'événement *E* : `occurrenceE.up()`

Règle de conception

- Identifier les événements qui doivent être attendus avant chaque action
- Définir un sémaphore *semE* par événement *E* à attendre
  - appel à *semE.down()* avant l'action où l'attente est nécessaire
  - appel à *semE.up()* après l'action provoquant l'occurrence de l'événement

Schémas d'utilisation essentiels (3/4)

Synchronisation élémentaire : rendez-vous entre 2 processus *A* et *B*

Problème : garantir l'exécution ≪ virtuellement ≫ simultanée d'un point donné du flot de contrôle de *A* et d'un point donné du flot de contrôle de *B*

- Objets partagés :

```
aArrivé = new Semaphore(0);
bArrivé = new Semaphore(0)  // initialisés à 0
```

- Protocole de rendez-vous :

```
Processus A          Processus B
  ...                  ...
aArrivé.up()         bArrivé.up()
bArrivé.down()       aArrivé.down()
  ...                  ...
```

Schémas d'utilisation essentiels (4/4)

Généralisation : rendez-vous à N processus (≪ barrière ≫)

Fonctionnement : pour passer la barrière, un processus doit attendre que les $N-1$ autres processus l'aient atteint.

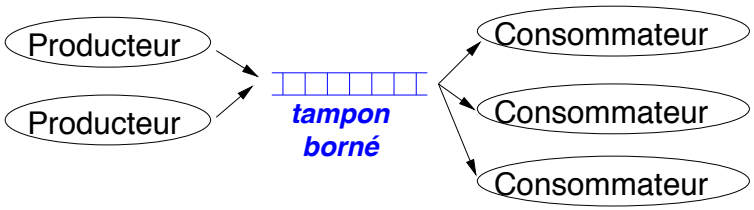
- Objet partagé :

```
barrière = tableau [0..N-1] de Semaphore;
pour i  := 0 à N-1 faire barrière[i].init(0) finpour;
```

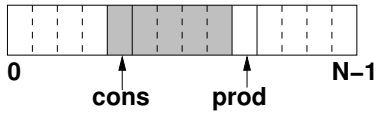
- Protocole de passage de la barrière (pour le processus *i*) :

```
pour k := 0 à N-1 faire
   barrière[i].up()
finpour;
pour k := 0 à N-1 faire
   barrière[k].down()
finpour;
```


Schéma producteurs/consommateurs : tampon borné



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs



producteur	*consommateur*
`produire(i) {i : Item}`	`occupé.down()`
`libre.down()`	*{ ∃ places occupées }*
{ ∃ places libres }	`mutex.down()`
`mutex.down()`	`{ retrait du tampon }`
`{ dépôt dans le tampon }`	`i := tampon[cons]`
`tampon[prod] := i`	`cons := cons + 1 mod N`
`prod := prod + 1 mod N`	`mutex.up()`
`mutex.up()`	*{ ∃ places libres }*
{ ∃ places occupées }	`libre.up()`
`occupé.up()`	`consommer(i) {i : Item}`

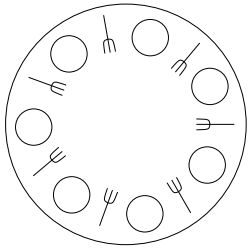
Sémaphores : `mutex` := 1, `occupé` := 0, `libre` := N

Contrôle fin du partage (1/3) : pool de ressources

- N ressources critiques, équivalentes, réutilisables
- usage exclusif des ressources
- opération `allouer` $k \leq N$ ressources
- opération `libérer` des ressources précédemment obtenues
- bon comportement :
  - pas deux demandes d'allocation consécutives sans libération intermédiaire
  - un processus ne libère pas plus que ce qu'il détient

Mise en œuvre de politiques d'allocation : FIFO, priorités...

Contrôle fin du partage (2/3) : philosophes et spaghettis



N philosophes sont autour d'une table.
Il y a une assiette par philosophe, et une fourchette entre chaque assiette.
Pour manger, un philosophe doit utiliser les deux fourchettes adjacentes à son assiette (et celles-là seulement).

Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Allocation multiple de ressources différenciées, interblocage...

Contrôle fin du partage (3/3) : lecteurs/rédacteurs

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.

Stratégies d'allocation pour des classes distinctes de clients ...

Plan

Implantation d'un sémaphore

Repose sur un service de gestion des processus fournissant :

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

Implantation

Sémaphore = ⟨ int nbjetons ;
File<Processus> bloqués ⟩

Algorithme

```
S.down() =  entrer en excl. mutuelle
              si S.nbjetons = 0 alors
                 insérer self dans S.bloqués
                 suspendre le processus courant
              sinon
                 S.nbjetons ← S.nbjetons - 1
              finsi
            sortir d'excl. mutuelle
---------------------------------------------
S.up() =  entrer en excl. mutuelle
            si S.bloqués ≠ vide alors
               procRéveillé ← extraire de S.bloqués
               débloquer procRéveillé
            sinon
               S.nbjetons ← S.nbjetons + 1
            finsi
          sortir d'excl. mutuelle
```


Compléments (1/3) :

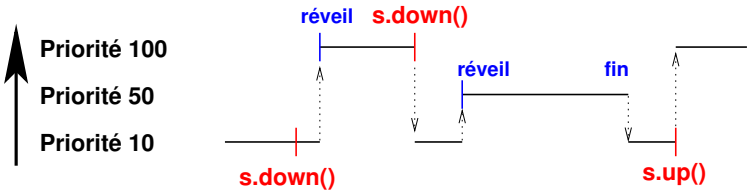
réalisation d'un sémaphore général à partir de sémaphores binaires

```
Sg = ⟨val := ?,
      mutex = new SemaphoreBinaire(1),
      accès = new SemaphoreBinaire(val>0;1;0) // verrous
      ⟩
Sg.down() =  Sg.accès.down()
             Sg.mutex.down()
             S.val ← S.val - 1
             si S.val ≥ 1 alors Sg.accès.up()
             Sg.mutex.up()
Sg.up() =  Sg.mutex.down()
           S.val ← S.val + 1
           si S.val = 1 alors Sg.accès.up()
           Sg.mutex.up()
```

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

Compléments (2/3) : sémaphores et priorités

Temps-réel ⇒ priorité ⇒ sémaphore non-FIFO.
Inversion de priorités : un processus moins prioritaire bloque/retarde indirectement un processus plus prioritaire.



Compléments (3/3) : solution à l'inversion de priorité

- Plafonnement de priorité (priority ceiling) : monter systématiquement la priorité d'un processus verrouilleur à la priorité maximale des processus potentiellement utilisateurs de cette ressource.
  - – Nécessite de connaître a priori les demandeurs
  - – Augmente la priorité même en l'absence de conflit
  - \+ Simple et facile à implanter
  - \+ Prédictible : la priorité est associée à la ressource
- Héritage de priorité : monter dynamiquement la priorité d'un processus verrouilleur à celle du demandeur.
  - \+ Limite les cas d'augmentation de priorité aux cas de conflit
  - – Nécessite de connaître les possesseurs d'un sémaphore
  - – Dynamique ⇒ comportement moins prédictible

Conclusion

Les sémaphores

- \+ ont une sémantique, un fonctionnement simples à comprendre
- \+ peuvent être mis en œuvre de manière efficace
- \+ sont suffisants pour réaliser les schémas de synchronisation nécessaires à la coordination des applications concurrentes
- − mais sont un outil de synchronisation élémentaire, aboutissant à des solutions difficiles à concevoir et à vérifier
  → schémas génériques