Sémaphores

— définition, applications, implémentation —

Introduction

Motivations pour des primitives de synchronisation de plus haut niveau :

- L'attente active est inefficace.
- Protocoles d'entrée et de sortie de section critique assez complexes.
- Mélange des variables utilisées par les protocoles d'entrée-sortie de section critique avec celles du programme.

Historique des sémaphores

- Dijkstra & co. 1968 avec le système d'exploitation THE
- la première primitive de synchronisation évoluée
- concept inspiré de la synchronisation ferroviaire

Type sémaphore général

Deux champs :

- `v` de type entier positif
- `l` de type ensemble de descripteurs de processus

Trois opérations :

- constructeur $(k, \emptyset)$: initialise `v` à $k \geq 0$ et `l` à $\emptyset$
- `wait()` : opération atomique d'acquisition

```
wait(S) { atomic { // p est le processus appelant
  if :: S.v > 0 -> S.v = S.v-1
     :: else  -> S.l = S.l U p
                 p.state = bloque
  fi } }
```

- `signal()` : opération atomique de libération

```
signal(S) { atomic { // p est le processus appelant
  if :: empty(S.l) -> S.v = S.v+1
     :: else  -> q = un element de l
                 S.l = S.l - q
                 q.state = pret
  fi } }
```


Type sémaphore binaire

Cas particulier intéressant : `v`$\in \{0, 1\}$.

Aussi appelé mutex.

Définition identique à celle du sémaphore général sauf pour :

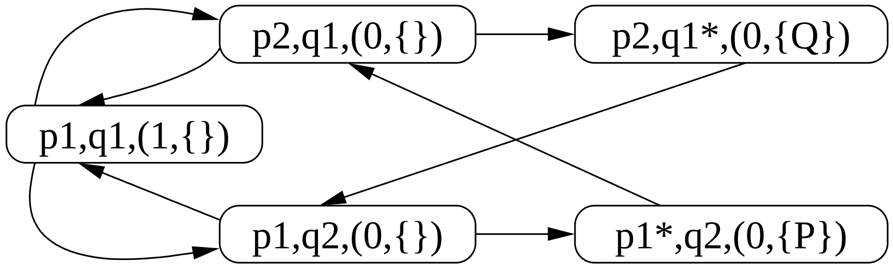
- `signal()` : opération atomique de libération

```
signal() { atomic { // p est le processus appelant
  if :: v == 1 -> // indefini
     :: empty(l) -> v = 1
     :: else  -> q = un element de l
                 l = l - q
                 q.state = pret
  fi
} }
```

Exemple : exclusion mutuelle binaire

```
    bsem S (1,emptyset);
    active proctype P() {
      do :: true ->
            /* section_non_critique */
p1:         wait(S);
            /* section critique */
p2:         signal(S)
      od
    }
    active proctype Q() {
    ...
    }
```

Le diagramme d'états est :



Ce qui prouve la correction de l'algorithme.

Invariants du sémaphore

Pour un sémaphore S, soient :

- k la valeur initiale du sémaphore
- $\#signal(S)$ le nombre d'opérations `signal` effectuées complètement
- $\#wait(S)$ le nombre d'opérations `wait` effectuées complètement. ;

Théorème (Invariants du sémaphore)
Un sémaphore S satisfait les invariants suivants :

$$S.v \geq 0 \quad (1)$$

$$S.v = k + \#signal(S) - \#wait(S) \quad (2)$$

Preuve : par induction sur les opérations possibles.
Initialement, $k \geq 0$ et $\#signal(S)$ et $\#wait(S)$ sont 0, donc les deux équations sont vraies.
Uniquement `wait` et `signal` changent les champs du sémaphore, il faut donc montrer que les deux opérations préservent les équations. Par exemple, si `wait` décrémente $S.v$, $\#wait(S)$ est incrémenté, ce qui conserve (2).

Théorème La solution avec sémaphore du problème de section critique est correcte, càd, avec exclusion mutuelle, sans inter-blocage et sans famine.

Preuve **(Exclusion mutuelle)**
Soit $\#CS$ le nombre de processus en section critique. On prouve par induction que :

$$\#CS = \#wait(S) - \#signal(S) \tag{3}$$

ce qui implique avec (2)

$$\#CS + S.v = 1 \tag{4}$$

donc, en utilisant (1), on a $\#CS \leq 1$.

(Absence d'interblocage)
Si les deux processus bloqués, $S.v = 0$ et $\#CS = 0$, ce qui contredit (4).

(Absence de famine)
Si $P[0]$ est en famine, il est bloqué dans $S.l$ et $S.v = 0$. Par (4), $\#CS = 1$ et donc $P[1]$ est en section critique. Par hypothèse de progrès en sec. critique, $P[1]$ doit sortir et exécuter `signal(S)`. Comme il n'y a que 2 processus, $S.l$ contient que $P[0]$ qui sera donc débloqué.

Exemple : exclusion mutuelle N-aire

Le même algorithme !!!

Prouver :

- exclusion mutuelle
- absence d'interblocage
- présence de famine

Pour quelles propriétés du sémaphore on évite la famine ?

Types de sémaphores

- faible : le champ $S.l$ est un ensemble quelconque
- fort : le champ $S.l$ est une file

```
wait(S) { atomic { // p est le processus appelant
  if :: S.v > 0 -> S.v = S.v-1
     :: else  -> S.l = append(S.l,p)
                 p.state = bloque
  fi } }
signal(S) { atomic { // p est le processus appelant
  if :: empty(S.l) -> S.v = S.v+1
     :: else  -> q = head(S.l)
                 S.l = tail(S.l)
                 q.state = pret
  fi } }
```

- avec attente : absence du champ $S.l$

```
wait(S) { atomic { S.v > 0 -> S.v=S.v-1 } }
signal(S) { atomic { S.v=S.v+1 } }
```

Uniquement le sémaphore fort assure l'absence de famine.

Sémaphores en Java

Classe `Semaphore` du `java.util.concurrent.Semaphore`.

Opérations :

- constructeurs `Semaphore(int k, bool f)` : initialise v avec k entier et f précise si équité
- `acquire()` = `wait()`
- `release()` = `signal()`

```
class Resources {
  private static final RES_NUMBER = 10;
  private final Semaphore token = new Semaphore(RES_NUMBER, true);
  private List resourcesPool = new ArrayList(RES_NUMBER);
  public Object getResource() throws InterruptedException {
    token.acquire();
    return resourcesPool.get();
  }
  public void releaseResource(Object o) {
    resourcesPool.put(o);
    token.release();
  }
}
```

Sémaphores en Promela

Facile à implémenter le sémaphore avec attente :

```
int S;
inline init(S,k) { S = k }
inline wait(S)   { atomic { S > 0 -> S=S-1 } }
inline signal(S) { atomic { S=S+1 } }
```

Exemple ...

Pour les autres types de sémaphores, il faut implémenter explicitement la liste avec un tableau.

Sémaphores
— applications —

Résumé du cours

1. Producteurs et consommateurs avec tampon unitaire : sémaphore binaire distribué
2. Producteurs et consommateurs avec tampon fini circulaire : comptage de ressources
3. Dîner des philosophes : synchronisation sur des ressources multiples
4. Lecteurs et écrivains : exclusion mutuelle sélective
5. Lecteurs et écrivains : technique du passage du rélais

Producteurs et consommateurs

Problème : les producteurs calculent et déposent au fur et à mesure la suite de leurs résultats ; les consommateurs prennent la suite des résultats déposés pour les utiliser.

Exemple : *pipe* Unix

Variantes du problème :

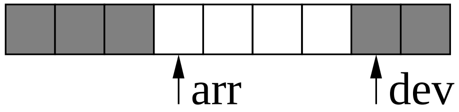
1. l'affichage est un tampon de taille unitaire (*1-place box*)
2. l'affichage est un tampon de taille finie
3. plusieurs consommateurs peuvent prendre un résultat dans le tampon
4. plusieurs producteurs peuvent déposer leurs résultats dans le tampon

```
/* type tampon */ buf;
proctype Prod(int i) { ...
  do :: true -> produit message m
                dépose le message m dans buf
  od }
proctype Cons(int j) { ...
  do :: true -> prend message m de buf
                consomme message m
  od }
```

Producteurs et consommateurs avec tampon 1 place

```
typeT buf;
bsem empty = 1; /* buf vide */
bsem full = 0;  /* buf plein */
/* (empty, full) = semaphore binaire distribue */
/* Invariant : 0 <= empty.v + full.v <= 1 */
proctype  Producer(int i) { ...
  do :: true ->
          produit message m
          depose: wait(empty);
                  buf = m;
                  signal(full)
  od
}
proctype  Consumer(int j) { ...
  do :: true ->
          prend: wait(full);
                 m = buf;
                 signal(empty)
          consomme message m
  od
}
```


Producteurs et consommateurs avec tampon fini circulaire



Un producteur et un consommateur

```
typeT buf[n];
int dev=0, arr=0;
sem empty = n, full = 0;
/* Invariant : n-2 <= empty + full <= n */
proctype  Producer() { ...
  do :: true -> produit message m
          depose: wait(empty);
                  buf[arr]=m; arr=(arr+1) mod n;
                  signal(full)
  od }
proctype  Consumer() { ...
  do :: true ->
          prend:  wait(full);
                  m=buf[dev]; dev=(dev+1) mod n;
                  signal(empty)
          consomme message m
  od }
```

Plusieurs producteurs et consommateurs

```
typeT buf[n];
int dev=0, arr=0;
sem empty = n, full = 0;
bsem mutexD = 1, mutexP = 1;
/* Invariant : n-2 <= empty + full <= n */
proctype  Producer(int i) { ...
  do :: true -> produit message m
          depose: wait(empty);
                  wait(mutexD);
                  buf[arr]=m; arr=(arr+1) mod n;
                  signal(mutexD);
                  signal(full)
  od
}
proctype  Consumer(int j) { ...
  do :: true ->
          prend: wait(full);
                 wait(mutexP);
                 m=buf[dev]; dev=(dev+1) mod n;
                 signal(mutexP);
                 signal(empty)
          consomme message m
  od
}
```


Dîner des philosophes

Problème : 5 philosophes sont assis autour d'une table ronde. Chaque philosophe passe sa vie entre penser et manger. Au centre de la table est toujours disponible un plat de spaghetti. Pour manger des spaghetti, chaque philosophe doit utiliser deux fourchettes. Malheureusement, il n'y a que cinq fourchettes, chaque fourchette étant placée entre deux philosophes. Les philosophes sont d'accord à utiliser que les fourchettes qui sont à leur gauche et droite immédiate.

Le problème est d'écrire un programme qui simule le comportement des philosophes et qui doit éviter que les philosophes meurent de faim par absence de fourchettes.

Exemple : accès à plusieurs variables partagées

```
proctype Philosophe(int i = 0 to 4) { ...
  do :: true -> pense
                obtient les fouchettes
                mange
                relâche les fourchettes
  od
}
```

Solution sans inter-blocage

Idée : casser la symétrie pour éviter l'inter-blocage.

```
bsem four[5] = { 1, 1, 1, 1, 1 };

proctype Philosophe(int i = 0 to 3) { ...
  do :: true -> pense
                wait(four[i]); wait(four[i+1]);
                mange
                signal(four[i]); signal(four[i+1])
  od
}

proctype Philosophe(int i = 4) { ...
  do :: true -> pense
                wait(four[(i+1) mod 5]); wait(four[i])
                mange
                signal(four[(i+1) mod 5]]); signal(fou
  od
}
```


Lecteurs et écrivains

Problème : Deux types de processus — lecteurs et écrivains — partagent une donnée (DB, fichier, liste, table, etc.). Les lecteurs exécutent des actions qui examinent la donnée ; les écrivains exécutent des actions qui examinent et modifient la donnée. La donnée est supposée être initialement dans un état consistent. Chaque action, si elle exécuté séparément, garde la consistance de la donnée. Afin d'éviter les interférences entre les actions, un écrivain doit avoir accès exclusif à la donnée. Si aucun écrivain n'accède la donnée, plusieurs lecteurs peuvent y accéder de façon concurrente.

C'est un problème d'exclusion mutuelle sélective, qui est une instace du problème d'allocation de ressources.

Peut-on concevoir des solution équitables pour toute type de processus ?

Solution avec sémaphores (préférence aux lecteurs)

```
int nr=0;       /* nombre de lecteurs */
bsem rw=1;      /* acces a la donnée */
bsem mutexR=1; /* acces a nr */
proctype Lecteur(int i) {
  do :: true -> ...
        wait(mutexR);
           nr=nr+1;
           if :: nr==1 -> wait(rw)
              :: else -> skip fi
        signal(mutexR);
        lecture donnée
        wait(mutexR);
           nr=nr-1;
           if :: nr==0 -> signal(rw)
              :: else -> skip fi
        signal(mutexR)
  od
}
proctype Ecrivain(int j) {
  do :: true -> ...
        wait(rw)
        lecture et ecriture donnée
        signal(rw)
  od
}
```


Technique du passage du rélais

Utilisation : implémentation avec des sémaphores des instructions atomiques conditionnelles d'un programme.

Soit :

```
I1: atomic { B1 -> S1 }
...
In: atomic { Bn -> Sn }
```

avec I1,...,In instructions d'un ou plusieurs processus ; Bi peut être vrai.

Implémentation avec sémaphores :

- e sémaphore binaire global, initialisé à 1 ; assure l'accès dans les sections critiques,
- bi sémaphore binaire associé à la garde Bi (si $\neq$ vrai), initialisé à 0 ; attente que Bi soit vrai.
- di entier associé à la garde Bi (si $\neq$ vrai), initialisé à 0 ; compte le nombre de processus bloqués sur Bi.

```
/* Ij: atomic { Sj } */
Ij: wait(e);
    Sj;
    SIGNAL

/* Ik: atomic { Bk -> Sk } */
Ik: wait(e);
    if :: !Bk -> dk=dk+1; signal(e); wait(bk)
       :: else -> skip fi;
    Sk;
    SIGNAL

SIGNAL:
  if :: B1 && d1>0 -> d1=d1-1; signal(b1)
     ...
     :: Bn && dn>0 -> dn=dn-1; signal(bn)
     :: else -> signal(e)
  fi
```


Exemple: Lecteurs et écrivains, solution non-primitive

```
int nr=0, nw=0;
{ RW: (nr==0 || nw==0) && nw <= 1 }

proctype Lecteur(int i) {
  do :: true -> ...

        atomic { nw==0 -> nr=nr+1 }


        lecture donnée

        atomic { nr=nr-1 }

  od }
proctype Ecrivain(int j) {
  do :: true -> ...

        atomic { (nr==0)&&(nw==0) -> nr=nr+1 }


        lecture et ecriture donnée

        atomic { nw=nw-1 }

  od }
```

Exemple: Lecteurs et écrivains, solution primitive

```
int nr=0, nw=0;
bsem e=1, r=0, w=0;
int dr=0, dw=0;
proctype Lecteur(int i) {
  do :: true -> ...
    wait(e);
    if :: !(nw==0) ->
            dr=dr+1; signal(e); wait(r)
       :: else -> skip fi;
    nr=nr+1;
    SIGNAL;
    lecture donnée
    wait(e);
    nr=nr-1;
    SIGNAL
  od
}
```


Exemple: Lecteurs et écrivains, solution primitive

```
proctype Ecrivain(int j) {
  do :: true -> ...
    wait(e);
    if :: !(nr==0)||!(nw==0) ->
            dw=dw+1; signal(e); wait(w)
       :: else -> skip fi;
    nw=nw+1;
    SIGNAL;
    lecture et ecriture donnée
    wait(e);
    nw=nw-1;
    SIGNAL
  od
}
```

Implémentation dans un noyau mono-processeur

On suppose qu'il existe :

- un type descripteur de sémaphore
- un tableau avec les descripteurs de sémaphores utilisés
- une primitive de création de sémaphore par sémaphore déclaré

```
create_sem(int init, int& nom) {
  obtient un descripteur de sémaphore libre
  initialise avec init sa valeur
  initialise nom avec la position du descripteur
  dispatcher()
}

wait(int nom) {
  trouve le descripteur de nom
  if (sa valeur > 0) valeur--
  else {
    insère exec_proc dans la file de nom
    exec_proc = 0
  }
```



```
  dispatcher()
}

signal(int nom) {
  trouve le descripteur de nom
  if (sa file est vide) valeur++
  else {
    enlève le processus en tête de liste
    insère ce processus à la fin de ready_proc
  }
  dispatcher()
}
```