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 mésure la suite de leurs résultats ; les consommateurs prennent la suite des résultats déposés pour les utiliser.

Exemple : *pipe* Unix

Variantes du problème :

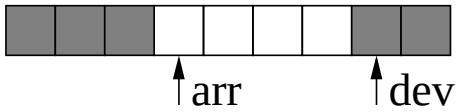
1. l'affichage est un tampon de taille unitaire (*1-place box*)
2. l'affichage est un tampon de taille finie
3. plusieurs consommateurs peuvent prendre un résultat dans le tampon
4. plusieurs producteurs peuvent déposer leurs résultats dans le tampon

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


Producteurs et consommateurs avec tampon unitaire

```
typeT buf;
bsem empty = 1; /* buf vide */
bsem full = 0;  /* buf plein */
/* (empty, full) = semaphore binaire distribue */
{ PC : 0 <= empty + full <= 1 }
proctype  Producer(int i) { ...
  do :: true ->
          produit message m
          depose: P(empty);
                  buf = m;
                  V(full)
  od
}
proctype  Consumer(int j) { ...
  do :: true ->
          prend: P(full);
                 m = buf;
                 V(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;
{ PC : n-2 <= empty + full <= n }
proctype  Producer() { ...
  do :: true -> produit message m
        depose: P(empty);
                buf[arr]=m; arr=(arr+1) mod n;
                V(full)
  od }
proctype  Consumer() { ...
  do :: true ->
        prend:  P(full);
                m=buf[dev]; dev=(dev+1) mod n;
                V(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;
{ PC : n-2 <= empty + full <= n }
proctype  Producer(int i) { ...
  do :: true -> produit message m
        depose: P(empty);
                P(mutexD);
                buf[arr]=m; arr=(arr+1) mod n;
                V(mutexD);
                V(full)
  od
}
proctype  Consumer(int j) { ...
  do :: true ->
        prend: P(full);
               P(mutexP);
               m=buf[dev]; dev=(dev+1) mod n;
               V(mutexP);
               V(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
                P(four[i]); P(four[i+1]);
                mange
                V(four[i]); V(four[i+1])
  od
}

proctype Philosophe(int i = 4) { ...
  do :: true -> pense
                P(four[(i+1) mod 5]); P(four[i]);
                mange
                V(four[(i+1) mod 5]]); V(four[i])
  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 -> ...
        P(mutexR);
           nr=nr+1;
           if :: nr==1 -> P(rw) fi
        V(mutexR);
        lecture donnée
        P(mutexR);
           nr=nr-1;
           if :: nr==0 -> V(rw) fi
        V(mutexR)
  od
}

proctype Ecrivain(int j) {
  do :: true -> ...
        P(rw)
        lecture et ecriture donnée
        V(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'accèes 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: P(e);
    Sj;
    SIGNAL

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

SIGNAL:
  if :: B1 && d1>0 -> d1=d1-1; V(b1)
     :: Bn && dn>0 -> dn=dn-1; V(bn)
     :: else -> V(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 -> ...
    P(e);
    if :: !(nw==0) -> dr=dr+1; V(e); P(r) fi;
    nr=nr+1;
    SIGNAL;
    lecture donnée
    P(e);
    nr=nr-1;
    SIGNAL
  od }
proctype Ecrivain(int j) {
  do :: true -> ...
    P(e);
    if :: !(nr==0)||!(nw==0) -> dw=dw+1; V(e); P(w) fi
    nw=nw+1;
    SIGNAL;
    lecture et ecriture donnée
    P(e);
    nw=nw-1;
    SIGNAL
  od }
```