Synchronisation des processus

Mécanismes

Gérard Padiou

Département Informatique et Mathématiques appliquées
ENSEEIHT

Septembre 2006

Plan

Concept proposé par Edsger W. Dijkstra

http ://fr.wikipedia.org/wiki/Edsger_Dijkstra

Définition d'un sémaphore

- Un sémaphore S encapsule un entier positif ou nul $S.Cnt \geq 0$ et une file d'attente de processus bloqués ;
- Deux opérations P et V permettent la mise-à-jour de l'entier ;
- L'opération P décrémente l'entier de 1 ssi $S.Cnt > 0$ $\Rightarrow$ processus appelant bloqué tant que $S.Cnt = 0$;
- L'opération V incrémente l'entier de 1 et éventuellement réveille un processus bloqué ($\equiv$ autorise une opération P).

Sémantique

$$\{S.Cnt = k \wedge k > 0\} \quad P(S) \quad \{S.Cnt = k - 1\}$$
$$\{S.Cnt = k\} \quad V(S) \quad \{S.Cnt = k + 1\}$$

Interprétation informelle : distributeur de tickets

- Valeur initiale du sémaphore v_0 = nombre de tickets
- P ≡ Prendre un ticket ;
- V ≡ Restituer un ticket.

Propriétés des sémaphores

- Sûreté : Mise-à-jour « cohérente » d'un compteur partagé ;
- Abstraction : Masquage de la gestion des files de processus ;
- Gestion de l'ordonnancement des processus à moyen terme.

Inconvénients

- Objet encapsulé et opérations immuables ;
- Programmation délicate.

Implantation du mécanisme de sémaphore

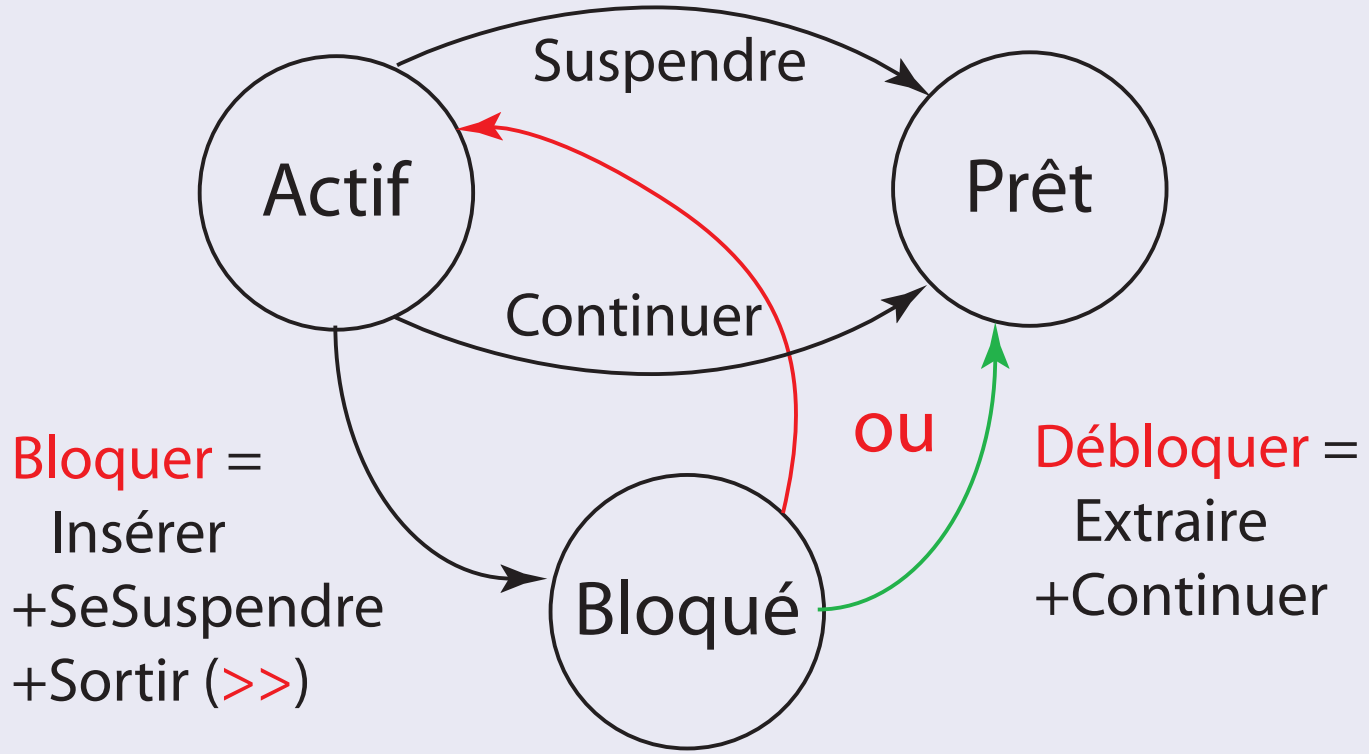
```
class Sémaphore {
 int Cnt ; /* compteur */
 Fifo<Processus> bloqués ; /* file de processus */

 P() { opération éventuellement bloquante
  <<
  Cnt-- ;
  if ( Cnt < 0) {
   bloqués.Insérer(Processus.getActif()) ;
   Processus.Suspendre() ; + sortir d'exclusion >>
  }
 >>
 }
```

Implantation du mécanisme de sémaphore (suite)

```
V() {
  <<
  Cnt++ ;
  if ( Cnt ≤ 0) {
    Processus réveillé = bloqués.Extraire() ;
    réveillé.Continuer() ;
  }
  >>
}
Sémaphore( int v0) {
  if ( v0 < 0 ) throw new Error("Valeur_Erronée") ;
  Cnt = v0 ; bloqués = new Fifo<Processus>() ;
}
} // class Sémaphore
```

Implantation du mécanisme de sémaphore

Graphe de transitions d'état



Utilisation du mécanisme de sémaphore

Hypothèses

- $\exists$ type (classe) sémaphore : Semaphore ;
- Création d'un objet : Semaphore S = new Semaphore(v_0)
- Appels : S.P() et S.V()

Travail d'analyse

- Déterminer les variables d'état du problème ;
- Pour chaque opération à synchroniser, préciser sa condition d'exécutabilité et son effet sur les variables d'état
- Encapsuler les variables d'états dans des sémaphores ;
- Traduire les opérations à synchroniser en terme de P ou V ;
- Problème majeur : pas toujours aussi simple (possible)...

Solution au problème de l'exclusion mutuelle

Analyse du problème

- Encapsuler le booléen indiquant l'état vis-à-vis de l'exclusion :
  $\longrightarrow$ booléen représentable par un entier : $\mathit{true} \equiv 1, \mathit{false} \equiv 0$
- $\{\mathit{libre}\}$ Entrer() $\{\neg \mathit{libre}\} \longrightarrow \{\mathrm{Cnt}{=}1\}$ S.P() $\{\mathrm{Cnt}{=}0\}$
- $\{\neg \mathit{libre}\}$Sortir() $\{\mathit{libre}\} \longrightarrow \{\mathrm{Cnt}{=}0\}$ S.V() $\{\mathrm{Cnt}{=}1\}$

Example

- Utiliser un sémaphore initialisé à 1 :

  Semaphore mutex = new Semaphore(1) ;
- Parenthéser les sections critiques par S.P(), S.V() :

  mutex.P() ; SC_i ; mutex.V() ;

Plan

Approche méthodologique

- Spécification du problème :
  - types de processus, de ressource(s) partagée(s),
  - interface des opérations du protocole de synchronisation,
  - propriétés de sûreté et vivacité à garantir.
- Modélisation du problème :
  - Définir l'interface d'accès aux ressources partagées : interface des opérations du protocole de synchronisation ;
  - Déterminer les variables d'états partagées : état des processus, compteurs, booléens, etc ;
  - Déterminer les gardes des opérations et leur effet sur l'état global partagé ;
- Utiliser un mécanisme générique pour assurer la synchronisation.

Approche méthodologique : cas des sémaphores

- **Cas simple** : encapsuler dans des sémaphores les variables d'état pour lesquelles les actions ont la sémantique des primitives P et V ;
- **Autre cas** : garantir l'exclusion mutuelle entre les opérations du protocole en utilisant un sémaphore d'exclusion mutuelle ;

ATTENTION : Eviter S.P() en section critique ($\Rightarrow$ interblocage).

Démarche possible

```
void op(...) {
  Mutex.P() ;
  si (op possible) alors S.V() ; ... ;
  Mutex.V()
  S.P() ; /* barrière */
}
```

Sémaphores particuliers

Sémaphores d'exclusion mutuelle (alias verrous)

- Sémaphore initialisé à la valeur 1 ;
- Utilisation restreinte : tout processus exécute une trace parenthésée d'opérations S.P() ; . . . ; S.V() sur le sémaphore.
- Usage : garantir l'exclusion mutuelle entre des sections de code accédant à des variables partagées ;

Sémaphores privés

- Sémaphore initialisé à la valeur 0 ;
- Un seul processus peut exécuter la primitive P ;
- Usage : couplage (événement $\rightarrow$ traitant) ;
- Implantation simple (Pourquoi ?).

Plan

Concept proposé par C.A.R. Hoare

http ://fr.wikipedia.org/wiki/Charles_Antony_Richard_Hoare

Définition d'un moniteur

- Moniteur ≡ Module encapsulant des données critiques ;
- Un moniteur est une ressource critique ;
- Tout accès aux données encapsulées ne peut se faire que par les procédures publiques du moniteur.

```
moniteur M {
   < données encapsulées >
   public void P1(...) { ... }
   ...
   public void Pn(...) { ... }
   ...
}
```

Synchronisation par variables conditions

Variable de type condition ≡ file d'attente de processus bloqués

- wait : blocage du processus appelant et insertion en file ;
- signal : extraction d'un processus et déblocage (s'il existe)

Règles d'ordonnancement

Lors d'un signal, deux processus candidats pour le moniteur :

- le signaleur
- le signalé (débloqué)

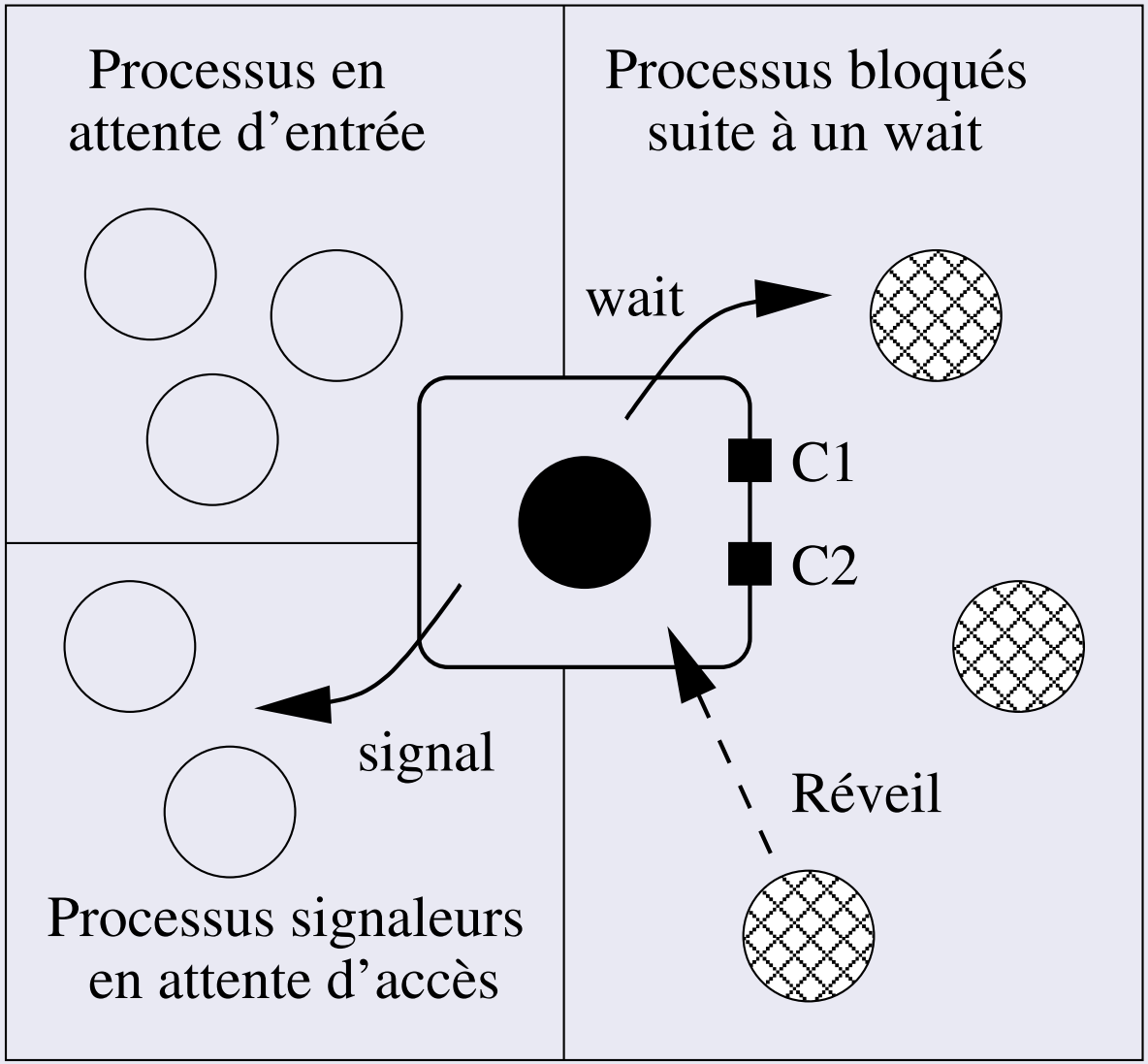
Problème : qui continue ?

Moniteurs de Hoare : priorité au signalé
Avantage : vérification de : pré(signal) ⇒ post(wait).
Inconvénient : conflit avec la priorité des processus

Implantation d'un sémaphore par un moniteur

```
moniteur Semaphore { /* invariant cnt ≥ 0 */
  int cnt ; condition positif ;
  public P() {
    if (cnt == 0) positif.wait() ;
    /* cnt > 0 */
    cnt-- ;
  }
  public V() {
    cnt++ ;
    /* cnt > 0 */ positif.signal() ;
  }
  Semaphore ( int v0) { cnt = v0 ; }
}
```

Ordonnancement des processus



Implantation d'un tampon partagé à N cases

```
moniteur TamponPartagé {
/* invariant nV ≥ 0 ∧ nP ≥ 0 ∧ nV + nP = N */
int nV ; int nP ;
condition nonPlein ; condition nonVide ;
int taille ; int t ; int q ; Message[] blocs ;
public Déposer(Message m) {
  if (nV == 0) nonPlein.wait() ;
  /* nV > 0 */
  nV-- ; blocs[q] = m ; q=(q+1)%taille ; nP++ ;
  /* nP > 0 */
  nonVide.signal() ;
}
```

suite...

```
public Message Retirer() {
  if (nP == 0) nonVide.wait() ;
  /* nP > 0 */
  nP-- ;
  Message me = blocs[t] ; t=(t+1)%taille ;
  nV++ ;
  /* nV > 0 */
  nonPlein.signal() ;
  return me ;
}
void TamponPartagé( int N ) {
  taille = N ; blocs = new Message[taille] ;
  nV = taille ; nP = 0 ; q = 0 ; t = 0 ;
}
} // moniteur TamponPartagé
```

En pratique...

Pas (plus) de langage avec le concept de moniteur MAIS

$\exists$ Mécanismes permettant de « simuler » (+ ou -) des moniteurs

Bibliothèque de mécanismes de synchronisation

- Mécanisme de verrou ($\equiv$ sémaphore d'exclusion mutuelle) :
  - Opérateurs lock/unlock,
  - Assure l'exclusion mutuelle d'usage du moniteur ;
- Mécanisme de condition :
  - Opérateurs wait/signal, wait/notify, wait/notifyAll, etc,
  - Sans garantie de priorité au signalé.
- Règle de programmation : remplacer les if par des while.

Exemple POSIX : les mécanismes proposés

Les verrous

- Type : pthread_mutex_t
- Création : pthread_mutex_init(pthread_mutex_t *mutex,...)
- int pthread_mutex_lock(pthread_mutext_t *mutex)
- int pthread_mutex_unlock(pthread_mutext_t *mutex)

Les variables conditions

- Type : pthread_cond_t
- Création : pthread_cond_init(pthread_cond_t *cond,...)
- int pthread_cond_wait(
  pthread_cond_t *cond, pthread_mutext_t *mutex)
- int pthread_cond_signal(pthread_cond_t *cond)

Programmation par aspect : fonctionnalité

```
class Tampon {
  Message blocs[] ;
  int taille ; int t = 0 ; int q = 0 ;
  void Déposer(Message Mes) {
    blocs[q] = Mes ; q = (q + 1) % taille ;
  }
  Message Retirer() {
    Message me = blocs[t] ; t = (t + 1) % taille ;
    return me ;
  }
  void Tampon(int N)
    { taille=N ; blocs = new Message[taille] ; }
}
```

Aspect « synchronisation »

```
coordinator of Tampon {
   invariant 0 <= #Deposer - #Retirer <= taille
   mutex Déposer, Retirer ;
   selfex Déposer, Retirer ;
      int #Deposer = 0 ; #Retirer = 0 ;
   guard Déposer :
      require #Déposer - #Retirer < taille ;
      onexit { #Déposer++ ; }
   guard Retirer :
      require #Déposer - #Retirer > 0 ;
      onexit { #Retirer++ ; }
}
```

Plan

Interface d'une tâche

Syntaxe :

```
task [type] <nom de tâche> is
   entry <nom d'entrée>[( <liste de paramètres> ) ] ;
   ...
end <nom de tâche> ;
```

Example

```
task [type] calculette is
   entry add( a,b in integer ; c out integer) ;
   entry sub( a,b in integer ; c out integer) ;
   ...
end calculette ;
```

Implantation d'une tâche

```
task body calculette is
  loop
   select
    accept add( a,b in integer ; c out integer)
     do c := a + b ; end ;
   or
    accept sub( a,b in integer ; c out integer)
     do c := a - b ; end ;
   ...
   end select ;
  end loop ;
end calculette ;
```

Appel avec délai de garde

```
select
    <référence de tâche>.<entrée>(...) ;
 or delay d ;
end select ;
```

ATTENTION : sémantique du select

Acceptation multiple

select
[when G_1=>] accept <e-1>(...) [do ...end <e-1> ;]
[<Bloc-1>]
...
or
[when G_i=>] accept <e-i>(...) [do ...end <e-i> ;]
[<Bloc-i>]
...
or
[when G_n=>] accept <e-n>(...) [do ...end<e-n> ;]
[<Bloc-n>]
end select

Notion de tâche de synchronisation

```
task synchroLR is   nl : INTEGER := 0;
  begin
   loop
    select
     when (debRed'count = 0) => accept debLec; nl := nl+1;
    or
     accept finLec; nl := nl - 1;
    or
     accept debRed do
      while (nl > 0) loop
       accept finLec; nl := nl - 1;
      end loop;
     end debRed;
     accept finRed;
    end select;
   end loop;
  end synchroLR;
```