

# Systèmes concurrents

2SN

ENSEEIH

Département Sciences du Numérique

8 octobre 2020

# Cinquième partie

## Moniteurs

## Contenu de cette partie

- motivation et présentation  
d'un objet de synchronisation « structuré » (moniteur)
- démarche de conception basée sur l'utilisation de moniteurs
- exemple récapitulatif (schéma producteurs/consommateurs)
- annexe : variantes et mise en œuvre des moniteurs

# Plan

## 1 Introduction

## 2 Définition

- Notion de moniteur Hoare, Brinch Hansen 1973
- Expression de la synchronisation : type « condition »
- Exemple
- Transfert du contrôle exclusif

## 3 Utilisation des moniteurs

- Méthodologie
- Exemple : producteurs/consommateurs

## 4 Conclusion

## 5 Annexes

- Allocateur de ressources
- Variantes
  - Réveil multiple
  - Priorité au signalé/signaleur
  - Régions critiques
- Implémentation des moniteurs par des sémaphores FIFO

# Limites des sémaphores

- imbrication aspects de synchronisation/aspects fonctionnels  
→ manque de modularité, code des processus interdépendant
- pas de contrainte sur le protocole d'utilisation des sémaphores  
→ démarche de conception artisanale, à partir de schémas élémentaires (attendre/signaler un événement, contrôler l'accès à une ressource. . . )
- approche (→ raisonnement) *opérateur*  
→ vérification difficile

## Exemples

- sections critiques entrelacées → interblocage
- attente infinie en entrée d'une section critique

# Plan

## 1 Introduction

## 2 Définition

- Notion de moniteur Hoare, Brinch Hansen 1973
- Expression de la synchronisation : type « condition »
- Exemple
- Transfert du contrôle exclusif

## 3 Utilisation des moniteurs

- Méthodologie
- Exemple : producteurs/consommateurs

## 4 Conclusion

## 5 Annexes

- Allocateur de ressources
- Variantes
  - Réveil multiple
  - Priorité au signalé/signaleur
  - Régions critiques
- Implémentation des moniteurs par des sémaphores FIFO

# Notion de moniteur Hoare, Brinch-Hansen 1973

## Idée de base

La synchronisation résulte du besoin de partager «convenablement» un objet entre plusieurs processus concurrents

- un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble de processus

## Définition

Un moniteur = un **module** exportant des **procédures** (*opérations*)

- Contrainte :  
exécution des procédures du moniteur en **exclusion mutuelle**
- La **synchronisation** des opérations du moniteur est réalisée par des **opérateurs internes au moniteur**.

Un moniteur est **passif** : ce sont les processus utilisant le moniteur qui l'activent, en invoquant ses procédures.

# Notion de moniteur Hoare, Brinch-Hansen 1973

## Idée de base

La synchronisation résulte du besoin de partager «convenablement» un objet entre plusieurs processus concurrents

- un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble de processus

## Définition

Un moniteur = un **module** exportant des **procédures** (*opérations*)

- Contrainte :  
exécution des procédures du moniteur en **exclusion mutuelle**
- La **synchronisation** des opérations du moniteur est réalisée par des **opérateurs internes au moniteur**.

Un moniteur est **passif** : ce sont les processus utilisant le moniteur qui l'activent, en invoquant ses procédures.





# Expression de la synchronisation : type *condition*

La *synchronisation* est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

- Une *file d'attente* est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition *C* :
  - *C.attendre()* [*C.wait()*] : bloque et range dans la file associée à *C* le processus appelant, puis libère l'accès exclusif au moniteur.
  - *C.signaler()* [*C.signal()*] : si des processus sont bloqués sur *C*, en réveille un ; sinon, nop (opération nulle).
- *condition*  $\approx$  événement
  - condition  $\neq$  sémaphore (pas de mémorisation des « signaux »)
  - condition  $\neq$  prédicat logique
- autres opérations sur les conditions :
  - *C.vide()* : renvoie vrai si aucun processus n'est bloqué sur *C*
  - *C.attendre(priorité)* : réveil des processus bloqués sur *C* selon une priorité

# Expression de la synchronisation : type *condition*

La *synchronisation* est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

- Une *file d'attente* est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition *C* :
  - *C.attendre()* [*C.wait()*] : bloque et range dans la file associée à *C* le processus appelant, puis libère l'accès exclusif au moniteur.
  - *C.signaler()* [*C.signal()*] : si des processus sont bloqués sur *C*, en réveille un ; sinon, nop (opération nulle).
- *condition*  $\approx$  événement
  - *condition*  $\neq$  sémaphore (pas de mémorisation des « signaux »)
  - *condition*  $\neq$  prédicat logique
- autres opérations sur les conditions :
  - *C.vide()* : renvoie vrai si aucun processus n'est bloqué sur *C*
  - *C.attendre(priorité)* : réveil des processus bloqués sur *C* selon une priorité

Exemple : travail délégué (schéma client/serveur asynchrone) : 1 client + 1 serveur

## Les activités (processus utilisant le moniteur)

### Client

*boucle*

⋮

déposer\_travail(t)

⋮

r ← lire\_résultat()

⋮

*fin\_boucle*

### Serveur

*boucle*

⋮

x ← prendre\_travail()

// (y ← f(x))

rendre\_résultat(y)

⋮

*fin\_boucle*

## Exemple – le moniteur

## Le moniteur

variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))

variables condition : Dépôt, Dispo

entrée déposer\_travail(in t)

{(pas d'attente)}

req ← t

Dépôt.signaler()

entrée prendre\_travail(out t)

si req = null alors

Dépôt.attendre()

finsi

t ← req

req ← null

{RAS}

entrée lire\_résultat(out r)

si rés = null alors

Dispo.attendre()

finsi

r ← rés

rés ← null

{RAS}

entrée rendre\_résultat(in y)

{(pas d'attente)}

rés ← y

Dispo.signaler()

nt

# Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.

→ Lors d'un **réveil** par *signaler()*, **qui** obtient l'accès exclusif ?

## Priorité au signalé

Lors du réveil par *signaler()*,

- l'accès exclusif est **transféré** au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

## Priorité au signaleur

Lors du réveil par *signaler()*,

- l'accès exclusif est **conservé** par le processus réveilleur ;
- le processus réveillé (signalé) est mis en attente
  - soit dans une file globale spécifique, prioritaire sur les processus entrants,
  - soit avec les processus entrants.

# Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.

→ Lors d'un **réveil** par *signaler()*, **qui** obtient l'accès exclusif ?

## Priorité au signalé

Lors du réveil par *signaler()*,

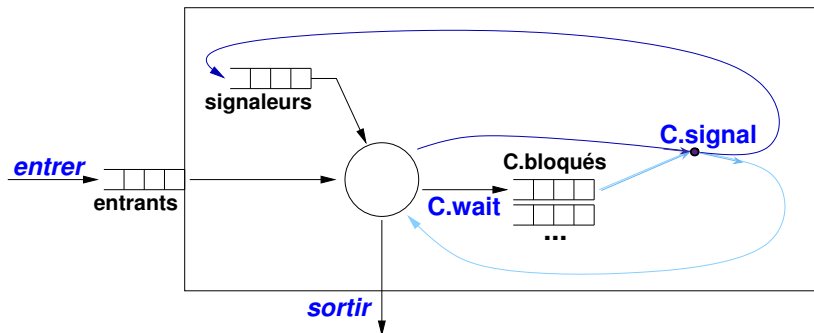
- l'accès exclusif est **transféré** au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

## Priorité au signaleur

Lors du réveil par *signaler()*,

- l'accès exclusif est **conservé** par le processus réveilleur ;
- le processus réveillé (signalé) est mis en attente
  - soit dans une file globale spécifique, prioritaire sur les processus entrants,
  - soit avec les processus entrants.

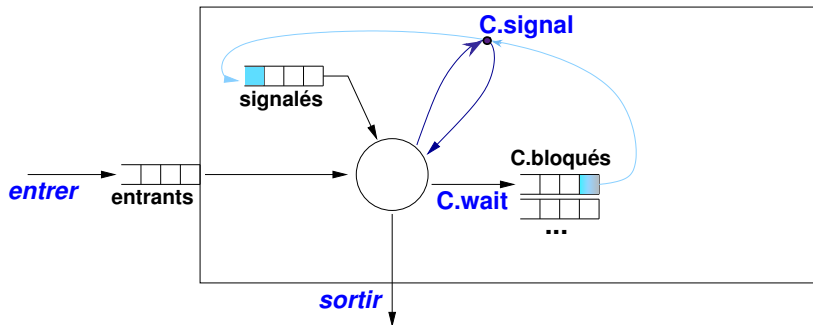
# Priorité au signalé



*C.signal()*

- = opération nulle si pas de bloqués sur *C*
- sinon,
  - suspend et ajoute le signaleur à la file des signaleurs
  - extrait le processus en tête des bloqués sur *C* et lui passe le contrôle
- signaleurs prioritaires sur les entrants (progression garantie)

# Priorité au signaleur **avec** file spécifique des signalés

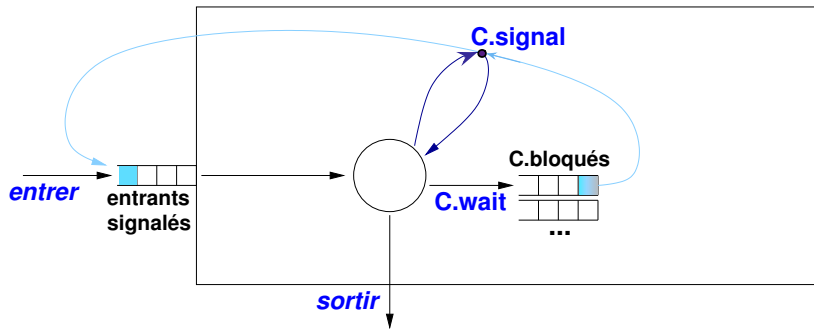


*C.signal()*

- si la file des bloqués sur *C* est non vide, en extrait le processus de tête et le range dans la file des signalés
- le signaleur conserve le contrôle
- signalés prioritaires sur les entrants



# Priorité au signaleur **sans** file spécifique des signalés



*C.signal()*

- si la file des bloqués sur *C* est non vide, en extrait le processus de tête et le range dans la file des entrants
- le signaleur conserve le contrôle
- signalés non prioritaires vis-à-vis des entrants

# Exemple signaleur vs signalé : travail délégué avec 1 client, 2 ouvriers

## Priorité au signalé

OK : quand un client dépose une requête et débloquent un ouvrier, celui-ci obtient immédiatement l'accès exclusif et prend la requête.

## Priorité au signaleur

- KO : situation : ouvrier n°1 bloqué sur `Dépôt.attendre()`.
- Le client appelle `déposer_travail` et en parallèle, l'ouvrier n°2 appelle `prendre_travail`. L'ouvrier n°2 attend l'accès exclusif.
- Lors de `Dépôt.signaler()`, l'ouvrier n°1 est débloquent de la var. condition et se met en attente de l'accès exclusif.
- Quand le client libère l'accès exclusif, qui l'obtient ? Si ouvrier n°2, il « vole » la requête, puis ouvrier n°1 obtient l'accès exclusif et récupère `null`.

# Comparaison des stratégies de transfert du contrôle

- **Priorité au signalé** : garantit que le processus réveillé obtient l'accès au moniteur **dans l'état où il était lors du signal**.
  - Raisonnement simplifié (le signaleur produit un état, directement utilisé par le signalé)
  - Absence de famine facilitée
- **Priorité au signaleur** : le réveillé obtient le moniteur **ultérieurement**, éventuellement après d'autres processus
  - Implantation du mécanisme plus simple et plus performante
  - Au réveil, le signalé doit **retester la condition de déblocage**
    - Possibilité de famine, écriture et raisonnements plus lourds

Peut-on simplifier encore l'expression de la synchronisation ?

# Peut-on simplifier encore l'expression de la synchronisation ?

## Idée (d'origine)

Attente sur des **prédicats**,  
 plutôt que sur des événements (= variables de type condition)  
 → opération unique : *attendre*(*B*), *B* expression booléenne

## Exemple : moniteur pour le tampon borné, avec *attendre*(prédicat)

variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))

entrée déposer\_travail(in t)  
 req ← t

entrée prendre\_travail(out t)  
 attendre(req ≠ null)  
 t ← req  
 req ← null

entrée lire\_résultat(out r)  
 attendre(rés ≠ null)  
 r ← rés  
 rés ← null

entrée rendre\_résultat(in y)  
 rés ← y

Pourquoi *attendre*(prédicat) n'est-elle pas disponible en pratique ?

### Efficacité problématique :

⇒ à chaque nouvel état (= à **chaque** affectation),  
évaluer **chacun** des prédicats attendus.

### → gestion de l'évaluation laissée au programmeur

- à chaque prédicat attendu ( $P$ )  
est associée une variable de type condition ( $P\_valide$ )
- *attendre*( $P$ ) est implantée par  
**si**  $\neg P$  **alors**  $P\_valide.attendre()$  **fsi**  $\{P\}$
- le programmeur a la possibilité de signaler ( $P\_valide.signaler()$ )  
les instants/états (**pertinents**) où  $P$  est valide

### Principe

- concevoir en termes de prédicats attendus, puis
- simuler cette attente de prédicats au moyen de variables de type condition

## Exemple – le moniteur (reprise planche 10)

### Le moniteur

variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))

variables condition : Dépôt, Dispo

*entrée déposer\_travail(in t)*

*{(pas d'attente)}*

req ← t

Dépôt.*signaler()*

*entrée prendre\_travail(out t)*

*si req = null alors*

Dépôt.*attendre()*

*finsi*

t ← req

req ← null

*{RAS}*

*entrée lire\_résultat(out r)*

*si rés = null alors*

Dispo.*attendre()*

*finsi*

r ← rés

rés ← null

*{RAS}*

*entrée rendre\_résultat(in y)*

*{(pas d'attente)}*

rés ← y

Dispo.*signaler()*

nt

# Plan

- 1 Introduction
- 2 Définition
  - Notion de moniteur Hoare, Brinch Hansen 1973
  - Expression de la synchronisation : type « condition »
  - Exemple
  - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
  - Méthodologie
  - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
  - Allocateur de ressources
  - Variantes
    - Réveil multiple
    - Priorité au signalé/signaleur
    - Régions critiques
  - Implémentation des moniteurs par des sémaphores FIFO



# Méthodologie (1/3) : motivation

Moniteur = réalisation (et gestion) d'un objet partagé

- permet de concevoir la synchronisation en termes d'interactions entre chaque processus et **un** objet partagé :  
les seules interactions autorisées sont celles qui laissent l'objet partagé dans un état cohérent
- **Invariant du moniteur** = ensemble des états possibles pour l'objet géré par le moniteur

Schéma générique : exécution d'une action  $A$  sur un objet partagé, caractérisé par un invariant  $I$

- ① *si* l'exécution de  $A$  (depuis l'état courant) invalide  $I$   
*alors attendre() finsi* { **prédicat d'acceptation** de  $A$  }
- ② effectuer  $A$  { → **nouvel état courant**  $E$  }
- ③ *réveiller()* les processus qui peuvent progresser à partir de  $E$

# Méthodologie (2/3)

## Etapes

- ➊ Déterminer l'**interface** du moniteur
- ➋ Énoncer en français les **prédicats d'acceptation** de chaque opération
- ➌ Dédire les **variables d'état**  
qui permettent d'écrire ces prédicats d'acceptation
- ➍ Formuler l'**invariant** du moniteur et les prédicats d'acceptation
- ➎ Associer à chaque prédicat d'acceptation une **variable condition** qui  
permettra d'attendre/signaler la validité du prédicat
- ➏ **Programmer** les opérations, en suivant le protocole générique  
précédent
- ➐ **Vérifier** que
  - l'invariant est vrai chaque fois que le contrôle du moniteur est transféré
  - les réveils ont lieu quand le prédicat d'acceptation est vrai

# Méthodologie (3/3)

## Structure standard d'une opération

*si* le prédicat d'acceptation est faux *alors*  
    *attendre()* sur la variable condition associée

*finsi*

{ (1) État nécessaire au bon déroulement }

Mise à jour de l'état du moniteur (action)

{ (2) État garanti (résultat de l'action) }

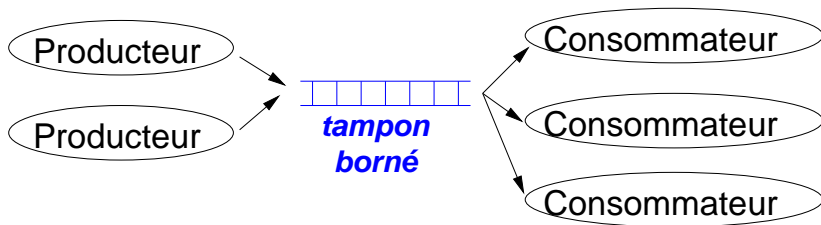
*signaler()* les variables conditions dont le prédicat associé est vrai

Vérifier, pour chaque variable condition, que

chaque précondition de *signaler()* (2)

implique chaque postcondition de *attendre()* (1)

## Exemple : réalisation du schéma producteurs/consommateurs



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

## 1 Interface :

- déposer(in v)
- retirer(out v)

## 2 Prédicats d'acceptation :

- déposer : il y a de la place, le tampon n'est pas plein
- retirer : il y a quelque chose, le tampon n'est pas vide

## 3 Variables d'état :

- nbOccupées : natural
- déposer :  $\text{nbOccupées} < N$
- retirer :  $\text{nbOccupées} > 0$

## 4 Invariant : $0 \leq \text{nbOccupées} \leq N$

## 5 Variables conditions : PasPlein, PasVide

## 1 Interface :

- déposer(in v)
- retirer(out v)

## 2 Prédicats d'acceptation :

- déposer : il y a de la place, le tampon n'est pas plein
- retirer : il y a quelque chose, le tampon n'est pas vide

## 3 Variables d'état :

- nbOccupées : natural
- déposer :  $\text{nbOccupées} < N$
- retirer :  $\text{nbOccupées} > 0$

## 4 Invariant : $0 \leq \text{nbOccupées} \leq N$

## 5 Variables conditions : PasPlein, PasVide

- ❶ Interface :
  - déposer(in v)
  - retirer(out v)
- ❷ Prédicats d'acceptation :
  - déposer : il y a de la place, le tampon n'est pas plein
  - retirer : il y a quelque chose, le tampon n'est pas vide
- ❸ Variables d'état :
  - nbOccupées : natural
  - déposer :  $\text{nbOccupées} < N$
  - retirer :  $\text{nbOccupées} > 0$
- ❹ Invariant :  $0 \leq \text{nbOccupées} \leq N$
- ❺ Variables conditions : PasPlein, PasVide

- ① Interface :
  - déposer(in v)
  - retirer(out v)
- ② Prédicats d'acceptation :
  - déposer : il y a de la place, le tampon n'est pas plein
  - retirer : il y a quelque chose, le tampon n'est pas vide
- ③ Variables d'état :
  - nbOccupées : natural
  - déposer :  $\text{nbOccupées} < N$
  - retirer :  $\text{nbOccupées} > 0$
- ④ Invariant :  $0 \leq \text{nbOccupées} \leq N$
- ⑤ Variables conditions : PasPlein, PasVide



- ❶ Interface :
  - déposer(in v)
  - retirer(out v)
- ❷ Prédicats d'acceptation :
  - déposer : il y a de la place, le tampon n'est pas plein
  - retirer : il y a quelque chose, le tampon n'est pas vide
- ❸ Variables d'état :
  - nbOccupées : natural
  - déposer :  $\text{nbOccupées} < N$
  - retirer :  $\text{nbOccupées} > 0$
- ❹ Invariant :  $0 \leq \text{nbOccupées} \leq N$
- ❺ Variables conditions : PasPlein, PasVide

- ① Interface :
  - déposer(in v)
  - retirer(out v)
- ② Prédicats d'acceptation :
  - déposer : il y a de la place, le tampon n'est pas plein
  - retirer : il y a quelque chose, le tampon n'est pas vide
- ③ Variables d'état :
  - nbOccupées : natural
  - déposer :  $\text{nbOccupées} < N$
  - retirer :  $\text{nbOccupées} > 0$
- ④ Invariant :  $0 \leq \text{nbOccupées} \leq N$
- ⑤ Variables conditions : PasPlein, PasVide

## déposer(in v)

```
si  $\neg(\text{nbOccupées} < N)$  alors
    PasPlein.attendre()
finsi
{ (1)  $\text{nbOccupées} < N$  }
// action applicative (ranger v dans le tampon)
nbOccupées ++
{ (2)  $N \geq \text{nbOccupées} > 0$  }
PasVide.signaler()
```

## retirer(out v)

```
si  $\neg(\text{nbOccupées} > 0)$  alors
    PasVide.attendre()
finsi
{ (3)  $\text{nbOccupées} > 0$  }
// action applicative (prendre v dans le tampon)
nbOccupées --
{ (4)  $0 \leq \text{nbOccupées} < N$  }
PasPlein.signaler()
```

# Vérification & Priorité

- Vérification :  $(2) \Rightarrow (3) ? (4) \Rightarrow (1) ?$
- Si priorité au signaleur, transformer si en tant que :

déposer(in v)

**tant que**  $\neg(\text{nbOccupées} < N)$  faire

PasPlein.wait

fintq

{ (1)  $\text{nbOccupées} < N$  }

// action applicative (ranger v dans le tampon)

$\text{nbOccupées}++$

{ (2)  $N \geq \text{nbOccupées} > 0$  }

PasVide.signal

# Plan

- 1 Introduction
- 2 Définition
  - Notion de moniteur Hoare, Brinch Hansen 1973
  - Expression de la synchronisation : type « condition »
  - Exemple
  - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
  - Méthodologie
  - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
  - Allocateur de ressources
  - Variantes
    - Réveil multiple
    - Priorité au signalé/signaleur
    - Régions critiques
  - Implémentation des moniteurs par des sémaphores FIFO

# Conclusion

Un moniteur implante un objet partagé, et contrôle la bonne utilisation de cet objet

## Apports

- modularité et encapsulation.
- la synchronisation est localisée dans le moniteur →
  - raisonnement simplifié
  - meilleure lisibilité

## Limites

- dans le moniteur, la synchronisation reste mêlée aux aspects fonctionnels
- la sémantique des moniteurs est complexe
- l'exclusion mutuelle sur les opérations d'un moniteur facilite la conception, mais :
  - est une source potentielle d'interblocages (moniteurs imbriqués)
  - est une limite du point de vue de l'efficacité

# Plan

- 1 Introduction
- 2 Définition
  - Notion de moniteur Hoare, Brinch Hansen 1973
  - Expression de la synchronisation : type « condition »
  - Exemple
  - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
  - Méthodologie
  - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
  - Allocateur de ressources
  - Variantes
    - Réveil multiple
    - Priorité au signalé/signaleur
    - Régions critiques
  - Implémentation des moniteurs par des sémaphores FIFO

# Allocateur de ressources

- $N$  ressources équivalentes, une activité en demande  $p \in 1..N$  puis les libère.
- Bon comportement : pas deux demandes consécutives sans libération (cf interblocage).
- Difficulté : une libération peut débloquent 0, 1 ou plusieurs demandeurs selon le nombre de ressources rendues et attendues.



# Allocateur de ressources - méthodologie

## ❶ Interface :

- demander( $p$ : 1..N)
- libérer( $q$ : 1..N)

## ❷ Prédicats d'acceptation :

- demander( $p$ ) : il y a au moins  $p$  ressources libres
- retirer( $q$ ) : rien

## ❸ Variables d'état :

- nbDispo : natural
- demander( $p$ ) :  $\text{nbDispo} \geq p$
- libérer( $q$ ) : *true*

## ❹ Invariant : $0 \leq \text{nbDispo} \leq N$

## ❺ Variable condition : AssezDeRessources

# Allocateur de ressources - méthodologie

## ❶ Interface :

- demander( $p: 1..N$ )
- libérer( $q: 1..N$ )

## ❷ Prédicats d'acceptation :

- demander( $p$ ) : il y a au moins  $p$  ressources libres
- retirer( $q$ ) : rien

## ❸ Variables d'état :

- nbDispo : natural
- demander( $p$ ) :  $\text{nbDispo} \geq p$
- libérer( $q$ ) : *true*

## ❹ Invariant : $0 \leq \text{nbDispo} \leq N$

## ❺ Variable condition : AssezDeRessources

# Allocateur de ressources - méthodologie

- ❶ Interface :
  - demander( $p: 1..N$ )
  - libérer( $q: 1..N$ )
- ❷ Prédicats d'acceptation :
  - demander( $p$ ) : il y a au moins  $p$  ressources libres
  - retirer( $q$ ) : rien
- ❸ Variables d'état :
  - nbDispo : natural
  - demander( $p$ ) :  $\text{nbDispo} \geq p$
  - libérer( $q$ ) : *true*
- ❹ Invariant :  $0 \leq \text{nbDispo} \leq N$
- ❺ Variable condition : *AssezDeRessources*

# Allocateur de ressources - méthodologie

- ❶ Interface :
  - demander( $p: 1..N$ )
  - libérer( $q: 1..N$ )
- ❷ Prédicats d'acceptation :
  - demander( $p$ ) : il y a au moins  $p$  ressources libres
  - retirer( $q$ ) : rien
- ❸ Variables d'état :
  - nbDispo : natural
  - demander( $p$ ) :  $\text{nbDispo} \geq p$
  - libérer( $q$ ) : *true*
- ❹ Invariant :  $0 \leq \text{nbDispo} \leq N$
- ❺ Variable condition : *AssezDeRessources*

# Allocateur de ressources - méthodologie

- ❶ Interface :
  - demander( $p: 1..N$ )
  - libérer( $q: 1..N$ )
- ❷ Prédicats d'acceptation :
  - demander( $p$ ) : il y a au moins  $p$  ressources libres
  - retirer( $q$ ) : rien
- ❸ Variables d'état :
  - nbDispo : natural
  - demander( $p$ ) :  $\text{nbDispo} \geq p$
  - libérer( $q$ ) : *true*
- ❹ Invariant :  $0 \leq \text{nbDispo} \leq N$
- ❺ Variable condition : AssezDeRessources

# Allocateur de ressources - méthodologie

- ❶ Interface :
  - demander( $p: 1..N$ )
  - libérer( $q: 1..N$ )
- ❷ Prédicats d'acceptation :
  - demander( $p$ ) : il y a au moins  $p$  ressources libres
  - retirer( $q$ ) : rien
- ❸ Variables d'état :
  - nbDispo : natural
  - demander( $p$ ) :  $\text{nbDispo} \geq p$
  - libérer( $q$ ) : *true*
- ❹ Invariant :  $0 \leq \text{nbDispo} \leq N$
- ❺ Variable condition : AssezDeRessources

# Allocateur – opérations

demander( $p$ )

si  $\neg(\text{nbDispo} < p)$  alors

    AssezDeRessources.wait

finsi

nbDispo  $\leftarrow$  nbDispo  $- p$

libérer( $q$ )

nbDispo  $\leftarrow$  nbDispo  $+ p$

si c'est bon alors     -- comment le coder ?

    AssezDeRessources.signal

finsi

# Allocateur – opérations

## demander( $p$ )

```
si  $\neg(\text{nbDispo} < p)$  alors  
    demande  $\leftarrow p$   
    AssezDeRessources.wait  
    demande  $\leftarrow 0$   
finsi  
nbDispo  $\leftarrow \text{nbDispo} - p$ 
```

## libérer( $q$ )

```
nbDispo  $\leftarrow \text{nbDispo} + p$   
si  $\text{nbDispo} \geq \text{demande}$  alors  
    AssezDeRessources.signal  
finsi
```



# Allocateur – opérations

## demander( $p$ )

Et s'il y a plusieurs demandeurs ?

```
si  $\neg(\text{nbDispo} < p)$  alors  
    demande  $\leftarrow p$   
    AssezDeRessources.wait  
    demande  $\leftarrow 0$   
finsi  
nbDispo  $\leftarrow \text{nbDispo} - p$ 
```

## libérer( $q$ )

```
nbDispo  $\leftarrow \text{nbDispo} + p$   
si  $\text{nbDispo} \geq \text{demande}$  alors  
    AssezDeRessources.signal  
finsi
```

# Allocateur – opérations

## demander( $p$ )

```
si demande  $\neq$  0 alors -- il y a déjà un demandeur  $\rightarrow$  j'attends mon tour  
    Sas.wait  
finsi  
si  $\neg(\text{nbDispo} < p)$  alors  
    demande  $\leftarrow p$   
    AssezDeRessources.wait    -- au plus un bloqué ici  
    demande  $\leftarrow 0$   
finsi  
nbDispo  $\leftarrow \text{nbDispo} - p$   
Sas.signal    -- au suivant de demander
```

## libérer( $q$ )

```
nbDispo  $\leftarrow \text{nbDispo} + p$   
si nbDispo  $\geq$  demande alors  
    AssezDeRessources.signal  
finsi
```

Note : priorité au signaleur  $\Rightarrow$  transformer le premier “si” de demander en “tant que” (ça suffit ici).

## Variante : réveil multiple : signalAll/broadcast

**C.signalAll** (ou broadcast) : *toutes* les activités bloquées sur la variable condition *C* sont débloquées. Elles se mettent en attente de l'accès exclusif.

**Rarement utilisé à bon escient.** Une solution triviale à un problème de synchronisation est d'utiliser une *unique* variable condition Accès et d'écrire *toutes* les procédures du moniteur sous la forme :

```

tant que  $\neg$ (condition d'acceptation) faire
    Accès.wait
fintq
...
Accès.signalAll  -- battez-vous

```

Mauvaise idée ! (performance, prédictibilité)

# Réveil multiple : cour de récréation unisexe

- ① type genre  $\triangleq$  (Fille, Garçon)  
 $\text{inv}(g) \triangleq$  si  $g = \text{Fille}$  alors Garçon sinon Fille
- ② Interface : entrer(genre) / sortir(genre)
- ③ Prédicats : entrer : personne de l'autre sexe / sortir : –
- ④ Variables : nb(genre)
- ⑤ Invariant :  $\text{nb}(\text{Filles}) = 0 \vee \text{nb}(\text{Garçons}) = 0$
- ⑥ Variables condition : accès(genre)

<pre> ⑥ entrer(genre g)   si nb(inv(g)) ≠ 0 alors     accès(g).wait   finsi   nb(g)++         </pre>	<pre> sortir(genre g)   nb(g)--   si nb(g) = 0 alors     accès(inv(g)).signalAll   finsi         </pre>
--	---

(solution naïve : risque de famine si un genre se coalise pour avoir toujours un membre présent dans la cour)



# Priorité au signaleur : transformation systématique ?

Pour passer de priorité au signalé à priorité au signaleur, transformer « si CA » en « tant que CA » n'est correct que si la condition d'acceptation (à l'entrée) et la condition de déblocage (au réveil) sont identiques.

Exemple : évitement de la famine : variable attente(genre) pour compter les enfants en attente et ne pas accaparer la cour.

```
entrer(genre g)
  si nb(inv(g))  $\neq$  0  $\vee$  attente(inv(g))  $\geq$  4 alors
    attente(g)++
    accès(g).wait
    attente(g)--
  fin si
  nb(g)++
```

Interblocage possible avec priorité signaleur et « tant que » à la place du « si » → repenser la solution.

## Variante : régions critiques

- Éliminer les variables conditions et les appels explicites à signaler  $\Rightarrow$  déblocages calculés par le système.
- Exclusion mutuelle plus « fine », en listant les variables partagées effectivement utilisées.

```
region liste des variables utilisées  
when prédicat logique  
do code
```

- 1 Attente que le prédicat logique soit vrai
- 2 Le code est exécuté en exclusion mutuelle vis-à-vis des autres régions ayant (au moins) une variable commune
- 3 À la fin du code, évaluation automatique des prédicats logiques des régions pour débloquent éventuellement.

## Exemple

tampon : shared array 0..N-1 of msg;

nbOcc : shared int := 0;

retrait, dépôt : shared int := 0, 0;

déposer(m)

  region

    nbOcc, tampon, dépôt

  when

    nbOcc < N

  do

    tampon[dépôt] ← m

    dépôt ← dépôt + 1 % N

    nbOcc ← nbOcc + 1

end

retirer()

  region

    nbOcc, tampon, retrait

  when

    nbOcc > 0

  do

    Result ← tampon[retrait]

    retrait ← retrait + 1 % N

    nbOcc ← nbOcc - 1

end

# Implémentation des moniteurs par des sémaphores FIFO

Dans le cas où les *signaler()* sont toujours en fin d'opération

- Exclusion mutuelle sur l'exécution des opérations du moniteur
  - définir un sémaphore d'exclusion mutuelle : *mutex*
  - encadrer chaque opération par *mutex.P()* et *mutex.V()*
- Réalisation de la synchronisation par variables condition
  - définir un sémaphore *SemC* (initialisé à 0) pour chaque condition *C*
  - traduire *C.attendre()* par *SemC.P()*, et *C.signaler()* par *SemC.V()*
  - Difficulté : pas de mémoire pour les appels à *C.signaler()*
    - éviter d'exécuter *SemC.V()* si aucun processus n'attend
    - un compteur explicite par condition : *cptC*
      - Réalisation de *C.signaler()* :  
si *cptC > 0* alors *SemC.V()* sinon *mutex.V()* fsi
      - Réalisation de *C.attendre()* :  
*cptC ++*; *mutex.V()*; *SemC.P()*; *cptC --*;

Dans le cas général : ajout d'un compteur et d'un sémaphore pour les processus signaleurs, réveillé prioritairement par rapport à *mutex*