

Systèmes concurrents

Philippe Quéinnec

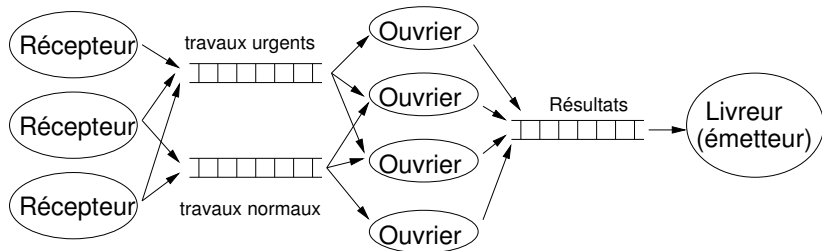
ENSEEIH
Département Sciences du Numérique

27 septembre 2024

Quatrième partie

Moniteurs

Motivation



- L'exclusion mutuelle résout les conflits d'accès aux files.
- Que fait un ouvrier s'il n'y a pas de travail à faire ?
- Comment un ouvrier peut-il *attendre* qu'une file soit non vide ?
- Comment un récepteur peut-il *signaler* qu'il a déposé un travail ?

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)
- Exemples avancés

Plan

- 1 Définition
 - Moniteur de Hoare, Brinch Hansen
 - Synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 2 Utilisation des moniteurs
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
- 3 Exemples avancés
 - Cours unisexe
 - Barrière
- 4 Conclusion

Moniteur de Hoare, Brinch-Hansen (1973)

Idée de base

La synchronisation résulte du besoin de partager convenablement un objet entre plusieurs activités concurrentes.

→ Un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par des activités.

Définition

Un moniteur = un **module** exportant des **opérations** (et éventuellement des constantes et des types).

- Contrainte :
exécution des opérations 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 activités utilisant le moniteur qui l'activent, en invoquant ses opérations.



Synchronisation : type *condition*

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

Définition

Variables de type **condition** définies dans le moniteur.

Opérations possibles sur une variable de type condition *C* :

- **C.wait()** (C.attendre()) : bloque l'activité appelante en libérant l'accès exclusif au moniteur.
- **C.signal()** (C.signaler()) : si des activités sont bloquées sur *C*, en réveille une; sinon, nop.
- Une **file d'attente** est associée à **chaque** variable condition
- **condition \approx événement**
→ condition \neq prédicat logique
- Autre opération sur les conditions :
C.empty() : vrai si aucune activité n'est bloquée sur *C*.

Exemple : travail délégué

Schéma client/serveur asynchrone : 1 client + 1 ouvrier

Les activités

Client

boucle

⋮

déposer_travail(t)

⋮

r ← lire_résultat()

⋮

fin_boucle

Ouvrier

boucle

⋮

x ← prendre_travail()

// (y ← f(x))

rendre_résultat(y)

⋮

fin_boucle

Exemple – le moniteur

variables d'état : req, rés

-- requête / résultat en attente (null si rien)

variables condition : Dépôt, Dispo

déposer_travail(*in* t)

{pas d'attente}

req ← t

Dépôt.alertsignal()

prendre_travail(*out* t)

si req = null alors

Dépôt.wait()

finsi

t ← req

req ← null

{RAS}

lire_résultat(*out* r)

si rés = null alors

Dispo.wait()

finsi

r ← rés

rés ← null

{RAS}

rendre_résultat(*in* y)

{pas d'attente}

rés ← y

Dispo.signal()

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.

→ Lors d'un **réveil** par `signal()`, **quelle activité** obtient/garde l'accès exclusif ?

Priorité au signalé

Lors du réveil par `signal()`,

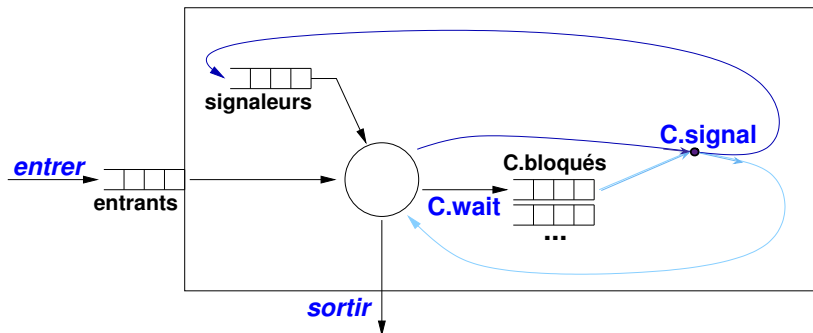
- l'accès exclusif est **transféré** à l'activité réveillée (signalée) ;
- l'activité signaleuse est mise en attente de pouvoir ré-acquérir l'accès exclusif.

Priorité au signaleur

Lors du réveil par `signal()`,

- l'accès exclusif est **conservé** par l'activité réveilleuse ;
- l'activité réveillée (signalée) est mise en attente de pouvoir acquérir l'accès exclusif
 - soit dans une file globale spécifique,
 - soit avec les activités entrantes.

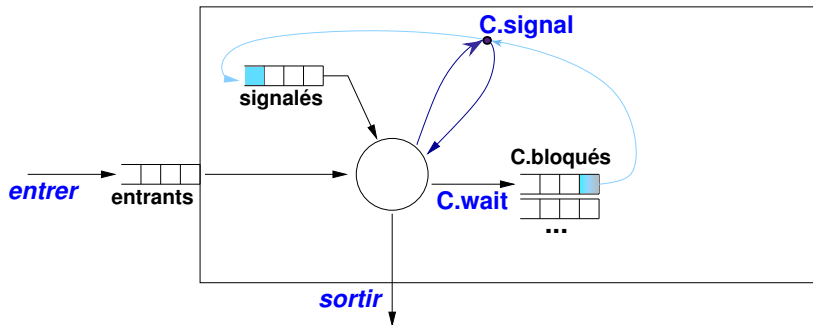
Priorité au signalé



C.signal()

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

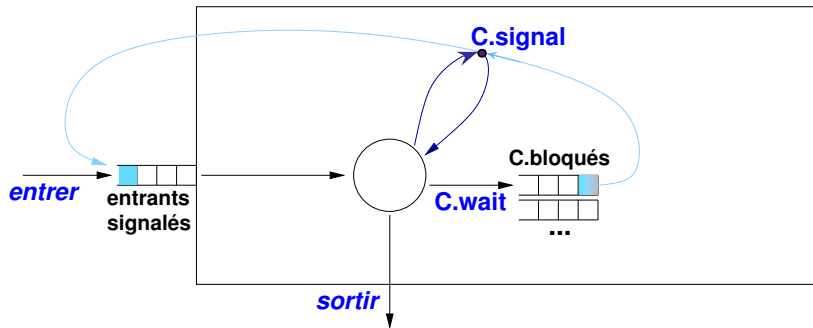
Priorité au signaleur avec file spécifique des signalés



C.signal()

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

Signaleur vs signalé : 1 client, 2 ouvriers

Priorité au signalé : OK

Quand un client dépose une requête et débloque un ouvrier, celui-ci obtient immédiatement l'accès exclusif et prend la requête.

Priorité au signaleur : KO

- Situation : ouvrier n°1 bloqué sur `Dépôt.wait()`.
- 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.signal()`, l'ouvrier n°1 est débloqué 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 l'activité réveillée 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 l'accès au moniteur **ultérieurement**, éventuellement après que d'autres activités ont eu accès au moniteur
 - Implantation plus simple et plus performante
 - Au réveil, le signalé doit **retester la condition de déblocage**

→ Possibilité de famine, écriture et raisonnements plus lourds

Exemple : Travail délégué avec 2 ouvriers et prio signaleur

variables d'état : req, rés

-- requête / résultat en attente (null si rien)

variables condition : Dépôt, Dispo

```
déposer_travail(in t)
{pas d'attente}
req ← t
Dépôt.signal()
```

```
prendre_travail(out t)
  tantque req = null faire
    Dépôt.wait()
  fintq
  t ← req
  req ← null
  {RAS}
```

```
lire_résultat(out r)
  tantque rés = null faire
    Dispo.wait()
  fintq
  r ← rés
  rés ← null
  {RAS}
```

```
rendre_résultat(in y)
{pas d'attente}
rés ← y
Dispo.signal()
```


Simplifier l'expression de la synchronisation ?

Idée d'origine

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

Exemple : travail délégué avec *wait*(prédicat)

variables d'état : req, rés

-- requête/résultat en attente (null si aucun(e))

déposer_travail(in t)

req ← t

prendre_travail(out t)

wait(req ≠ null)

t ← req

req ← null

lire_résultat(out r)

wait(rés ≠ null)

r ← rés

rés ← null

rendre_résultat(in y)

rés ← y

Pourquoi *wait*(prédicat) n'est-il 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

- une variable de type condition (*P_valide*) est associée à chaque prédicat attendu (*P*)
- *wait*(*P*) est implanté par
`si $\neg P$ alors P_valide.wait() fsi`
- le programmeur a la responsabilité de signaler (*P_valide.signal()*) les instants/états où *P* est valide

Principe

- Concevoir en termes de prédicats attendus
- Simuler cette attente de prédicats avec des variables conditions

Plan

- 1 Définition
 - Moniteur de Hoare, Brinch Hansen
 - Synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 2 Utilisation des moniteurs
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
- 3 Exemples avancés
 - Cours unisexe
 - Barrière
- 4 Conclusion

Méthodologie (1/3) : motivation

Moniteur = réalisation d'un objet partagé

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

Schéma générique d'une action sur l'objet partagé

- ❶ si l'exécution de A (depuis l'état courant) invalide I
alors attendre() finsi { **prédicat d'acceptation** de A }
- ❷ effectuer A { \rightarrow nouvel état courant E }
- ❸ signaler() les activités qui peuvent progresser à partir de E

Méthodologie (2/3)

Etapas

- ① Déterminer l'**interface** du moniteur
- ② Énoncer informellement les **prédicats d'acceptation** de chaque opération
- ③ Dédire les **variables d'état** qui permettent d'écrire ces prédicats d'acceptation
- ④ Formuler l'**invariant** du moniteur et les prédicats d'acceptation
- ⑤ Associer à chaque prédicat d'acceptation une **variable condition** qui permettra d'attendre/signaler la validité du prédicat
- ⑥ **Programmer** les opérations, en suivant le schéma précédent
- ⑦ **Vérifier** que
 - l'invariant est vrai à chaque transfert du contrôle du moniteur
 - le prédicat d'acceptation est vrai quand un réveil a lieu

Méthodologie (3/3)

Structure standard d'une opération

si le prédicat d'acceptation est faux alors

 Attendre (*wait*) sur la variable condition associée

finsi

{ (1) État nécessaire au bon déroulement }

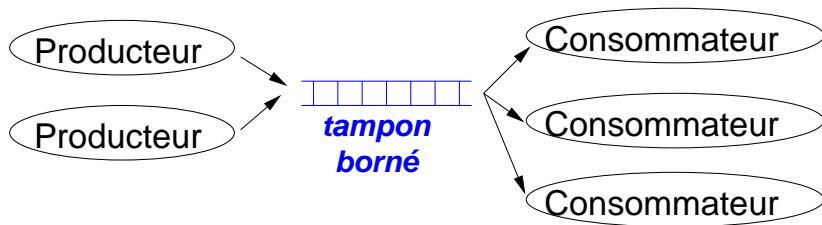
Mise à jour de l'état du moniteur (action)

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

Signaler (*signal*) 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).

Schéma producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs
- objectifs : ne pas écraser une case occupée, une unique lecture consommatrice par case, attendre pour déposer si plein, attendre pour retirer si vide

Méthodologie appliquée aux producteurs/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

Méthodologie appliquée aux producteurs/consommateurs

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

Méthodologie appliquée aux producteurs/consommateurs

- 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

Méthodologie appliquée aux producteurs/consommateurs

- 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

Méthodologie appliquée aux producteurs/consommateurs

- 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

Méthodologie appliquée aux producteurs/consommateurs

- 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

déposer(in v)

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

retirer(out v)

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

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

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. Le code d'une activité est décrit par :

boucle

$p \leftarrow \text{valeur} \in 1..N$

 demander(p)

 :

 libérer(p)

 :

finboucle

- Difficulté : une libération peut débloquent 0, 1 ou plusieurs demandeurs selon le nombre de ressources rendues et attendues.

Allocateur de ressources - méthodologie

1 Interface :

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

2 Prédicats d'acceptation :

- `demander(p)` : il y a au moins p ressources libres
- `libérer(q)` : rien

3 Variables d'état :

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

4 Invariant : $0 \leq \text{nbDispo} \leq N$

5 Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1 Interface :

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

2 Prédicats d'acceptation :

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

3 Variables d'état :

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

4 Invariant : $0 \leq \text{nbDispo} \leq N$

5 Variable condition : AssezDeRessources

Allocateur de ressources - méthodologie

1 Interface :

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

2 Prédicats d'acceptation :

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

3 Variables d'état :

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

4 Invariant : $0 \leq \text{nbDispo} \leq N$

5 Variable condition : AssezDeRessources

Allocateur de ressources - méthodologie

1 Interface :

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

2 Prédicats d'acceptation :

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

3 Variables d'état :

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

4 Invariant : $0 \leq \text{nbDispo} \leq N$

5 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
 - libérer(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
 - libérer(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 $\neg(\text{nbDispo} \geq p)$ alors

 AssezDeRessources.wait

finsi

nbDispo \leftarrow nbDispo $- p$

libérer(q)

nbDispo \leftarrow nbDispo $+ q$

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

 AssezDeRessources.signal

finsi

Allocateur – opérations

demander(p)

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

libérer(q)

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

Ajout de la variable d'état demande

Allocateur – opérations

demander(p)

Et s'il y a plusieurs demandeurs ?

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

libérer(q)

```
nbDispo  $\leftarrow \text{nbDispo} + q$   
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  
        Accès.wait  
    finsi  
    si  $\neg(\text{nbDispo} \geq p)$  alors  
        demande  $\leftarrow p$   
        AssezDeRessources.wait    -- au plus un bloqué ici  
        demande  $\leftarrow 0$   
    finsi  
    nbDispo  $\leftarrow \text{nbDispo} - p$   
    Accès.signal    -- au suivant de demander
```

libérer(q)

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

Note : dans le cas de moniteur avec priorité au signaleur \Rightarrow transformer le premier “si” de demander en “tant que” (suffit ici).

Bilan

- Moniteur = module / objet exportant des opérations
 - Exécution en exclusion mutuelle des opérations
(protection des variables internes du moniteur)
 - Variables conditions pour bloquer/débloquer
(à la charge du programmeur)
- Méthodologie de conception : prédicats d'acceptation → variables conditions + structure usuelle des opérations
- Difficulté : sémantique priorité au signaleur / signalé ?

Plan

- 1 Définition
 - Moniteur de Hoare, Brinch Hansen
 - Synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 2 Utilisation des moniteurs
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
- 3 Exemples avancés
 - Cours unisexe
 - Barrière
- 4 Conclusion

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 serait d'utiliser une *unique* variable condition
Accès et d'écrire *toutes* les opérations 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

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

<pre> 6 entrer(genre g) si nb(inv(g)) \neq 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)



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 genre / 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)



Réveil multiple : cour de récréation unisexe

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

<pre> 6 entrer(genre g) si nb(inv(g)) \neq 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)



Réveil multiple : cour de récréation unisexe

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

<pre> 6 entrer(genre g) si nb(inv(g)) \neq 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)



Réveil multiple : cour de récréation unisexe

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

6 entrer(genre g) si nb(inv(g)) \neq 0 alors accès(g).wait finsi nb(g)++	sortir(genre g) nb(g)-- si nb(g) = 0 alors accès(inv(g)).signalAll finsi
----------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------

(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.

Contre-exemple : éviter la famine : variable attente(genre) pour compter les enfants en attente et ne pas accaparer la cour.

```

entrer(genre g)
  si nb(inv(g)) ≠ 0 ∨ attente(inv(g)) ≥ 4 alors
    attente(g)++
    accès(g).wait
    attente(g)--
  finsi
  nb(g)++

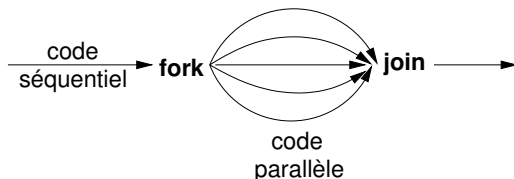
```

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

Barrière

- 1 Barrière élémentaire : ensemble d'activités qui attendent mutuellement qu'elles soient toutes au même point (rendez-vous multiple)
- 2 Barrière généralisée :
 - barrière de taille M alors qu'il existe N candidats ($N > M$)
 - barrière réutilisable (cyclique) : nécessité de la refermer

Schéma de
parallélisme
« fork-join »



Barrière à N activités - méthodologie

1 Interface :

- `franchir()`

2 Prédicats d'acceptation :

- `franchir()` : N activités ont demandé à franchir

3 Variables d'état :

- `nbArrivés` : natural
- `franchir()` : `nbArrivés = N`

4 Invariant : $0 \leq \text{nbArrivés} \leq N$

5 Variable condition : `BarrièreLevée`

Barrière à N activités - méthodologie

- ❶ Interface :
 - franchir()
- ❷ Prédicats d'acceptation :
 - franchir() : N activités ont demandé à franchir
- ❸ Variables d'état :
 - nbArrivés : natural
 - franchir() : $\text{nbArrivés} = N$
- ❹ Invariant : $0 \leq \text{nbArrivés} \leq N$
- ❺ Variable condition : BarrièreLevée

Barrière à N activités - méthodologie

- ❶ Interface :
 - franchir()
- ❷ Prédicats d'acceptation :
 - franchir() : N activités ont demandé à franchir
- ❸ Variables d'état :
 - nbArrivés : natural
 - franchir() : nbArrivés = N
- ❹ Invariant : $0 \leq \text{nbArrivés} \leq N$
- ❺ Variable condition : BarrièreLevée

Barrière à N activités - méthodologie

- ❶ Interface :
 - franchir()
- ❷ Prédicats d'acceptation :
 - franchir() : N activités ont demandé à franchir
- ❸ Variables d'état :
 - nbArrivés : natural
 - franchir() : $\text{nbArrivés} = N$
- ❹ Invariant : $0 \leq \text{nbArrivés} \leq N$
- ❺ Variable condition : `BarrièreLevée`

Barrière à N activités - méthodologie

- ❶ Interface :
 - franchir()
- ❷ Prédicats d'acceptation :
 - franchir() : N activités ont demandé à franchir
- ❸ Variables d'état :
 - nbArrivés : natural
 - franchir() : $\text{nbArrivés} = N$
- ❹ Invariant : $0 \leq \text{nbArrivés} \leq N$
- ❺ Variable condition : `BarrièreLevée`

Barrière à N activités - méthodologie

- ❶ Interface :
 - franchir()
- ❷ Prédicats d'acceptation :
 - franchir() : N activités ont demandé à franchir
- ❸ Variables d'état :
 - nbArrivés : natural
 - franchir() : $\text{nbArrivés} = N$
- ❹ Invariant : $0 \leq \text{nbArrivés} \leq N$
- ❺ Variable condition : BarrièreLevée

Barrière à N activités – opération

```
franchir()
```

```
nbArrivés++
```

```
si  $\neg(\text{nbArrivés} = N)$  alors
```

```
    BarrièreLevée.wait
```

```
finsi
```

```
{ nbArrivés =  $N$  }
```

```
BarrièreLevée.signal    // réveil en chaîne du suivant
```

```
nbArrivés--             // ou nbArrivés  $\leftarrow 0$ 
```

Note : On pourrait remplacer le réveil en chaîne par :

```
si nbArrivés=N alors BarrièreLevée.signalAll
```

(la sémantique de SignalAll en priorité au signalé est fragile : un seul obtient l'accès exclusif, les autres attendent leur tour)

Barrière à N activités – Priorité au signaleur ?

- Correct avec priorité au signalé
- **Incorrect** avec priorité au signaleur :
 - $\geq N$ peuvent passer :
Le n -ième arrive, signale, décrémente et libère l'accès exclusif ; pendant ce temps un $n+1$ -ième est arrivé ; s'il obtient l'accès exclusif avant celui signalé \Rightarrow il passe et signale ; etc. Puis tous ceux signalés passent.
 - Remplacement du si en tant que : un seul passe :
Le n -ième arrive, signale, décrémente et libère l'accès exclusif ; celui réveillé reteste la condition, trouve nbArrivés à $N - 1$ et se rebloque.

La condition de réveil (il y a eu N arrivées) est plus faible que la condition de passage (il y a actuellement N arrivées en attente). Retester la condition de passage est trop fort.

→ se souvenir que N activités sont en cours de franchissement.

Barrière à N activités – opération

franchir(), priorité au signalneur

```
tant que (nbArrivés =  $N$ ) alors
    // barrière en cours de vidage
    BarrièreBaissée.wait
fintq

nbArrivés++
tant que  $\neg$ (nbArrivés =  $N$ ) alors
    BarrièreLevée.wait
fintq
si nbArrivés =  $N$   $\wedge$  nbSortis = 0 alors // dernier arrivé
    BarrièreLevée.signalAll
finsi

nbSortis++
si nbSortis =  $N$  alors // dernier sorti
    nbSortis  $\leftarrow$  0
    nbArrivés  $\leftarrow$  0
    BarrièreBaissée.signalAll
finsi
```

Plan

- 1 Définition
 - Moniteur de Hoare, Brinch Hansen
 - Synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 2 Utilisation des moniteurs
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
- 3 Exemples avancés
 - Cours unisexe
 - Barrière
- 4 Conclusion

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 des opérations facilite la conception mais :
 - est une source d'interblocages (moniteurs imbriqués)
 - limite l'efficacité (pas de parallélisme)