

Systèmes concurrents

Philippe Quéinnec

5 septembre 2019



Cinquième partie

Moniteur



Contenu de cette partie

- Un objet de synchronisation structuré : le moniteur
- Démarche de conception basée sur l'utilisation de moniteurs
- Exemple récapitulatif (schéma producteurs/consommateurs)
- Variantes



Plan

- 1 Spécification
 - Définition
 - Transfert du contrôle exclusif
- 2 Applications
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
 - Barrière
- 3 Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Régions critiques



Limites des sémaphores

- Imbrication aspects de synchronisation/aspects fonctionnels
→ manque de modularité, code des activités 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 *opératoire*
→ raisonnement et vérification difficiles



Définition – Hoare 1974

Idée de base

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

Définition

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

- Contrainte d'**exclusion mutuelle** pour l'exécution des procédures du moniteur : au plus une procédure en cours d'exécution.
- Mécanisme de **synchronisation interne**.

Un moniteur est **passif** : ce sont les activités qui invoquent ses procédures.



Synchronisation : variable condition

Définition

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

Opérations possibles sur une condition C :

- **$C.wait$** : bloque l'activité appelante en libérant l'accès exclusif au moniteur.
- **$C.signal$** : s'il y a des activités bloquées sur C , en réveille une ; sinon, nop.

condition \approx événement

- condition \neq sémaphore (pas de mémorisation des « signaux »)
- condition \neq prédicat logique



Exemple élémentaire – travail délégué

Travail délégué : 1 client + 1 ouvrier

Les activités

Client

```
boucle
  :
  déposer_travail(t)
  :
  r ← lire_résultat()
  :
finboucle
```

Ouvrier

```
boucle
  :
  x ← prendre_travail()
  // (y ← f(x))
  rendre_résultat(y)
  :
finboucle
```

Exemple élémentaire – Le moniteur

variables d'état : req, rés

variables conditions : TravailDéposé, RésultatDispo

```
déposer_travail(in t)
  req ← t
  TravailDéposé.signal
```

```
prendre_travail(out t)
  si req = null alors
    TravailDéposé.wait
  finssi
  t ← req
  req ← null
```

```
lire_résultat(out r)
  si rés = null alors
    RésultatDispo.wait
  finssi
  r ← rés
  rés ← null
```

```
rendre_résultat(in y)
  rés ← y
  RésultatDispo.signal
```

Transfert du contrôle exclusif

Le code dans le moniteur est exécuté en exclusion mutuelle.
Lors d'un réveil par `signal`, qui obtient/garde l'accès exclusif ?

Priorité au signalé

Lors du réveil par `signal`,

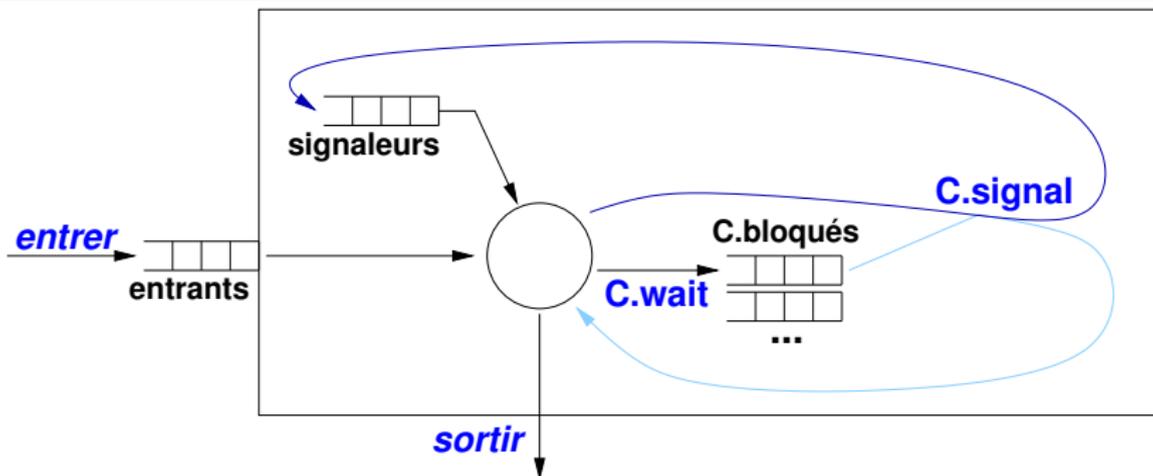
- l'accès exclusif est **transféré** à l'activité réveillée (signalée) ;
- l'activité signaleuse est mise en attente de pouvoir ré-acquérir l'accès exclusif.

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.

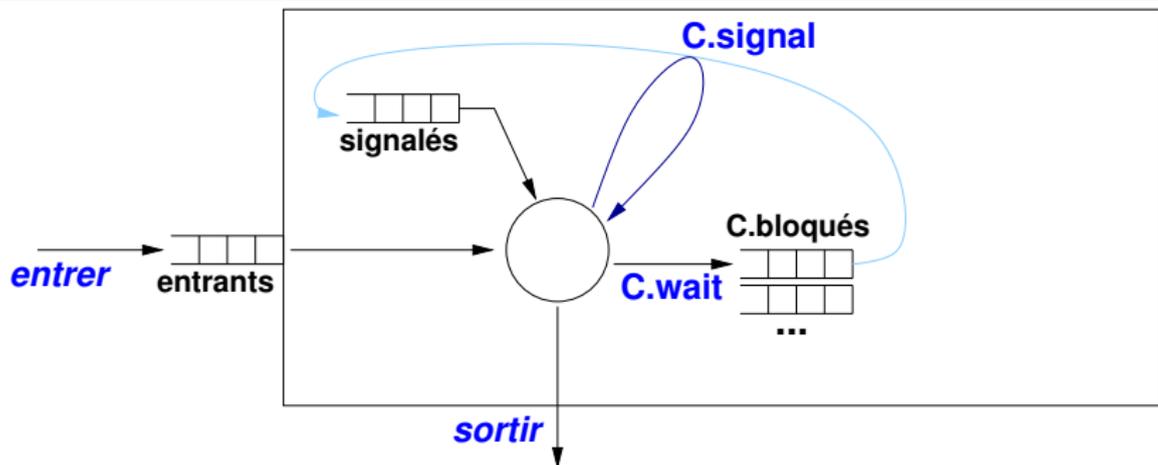
Priorité au signalé



C.signal()

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

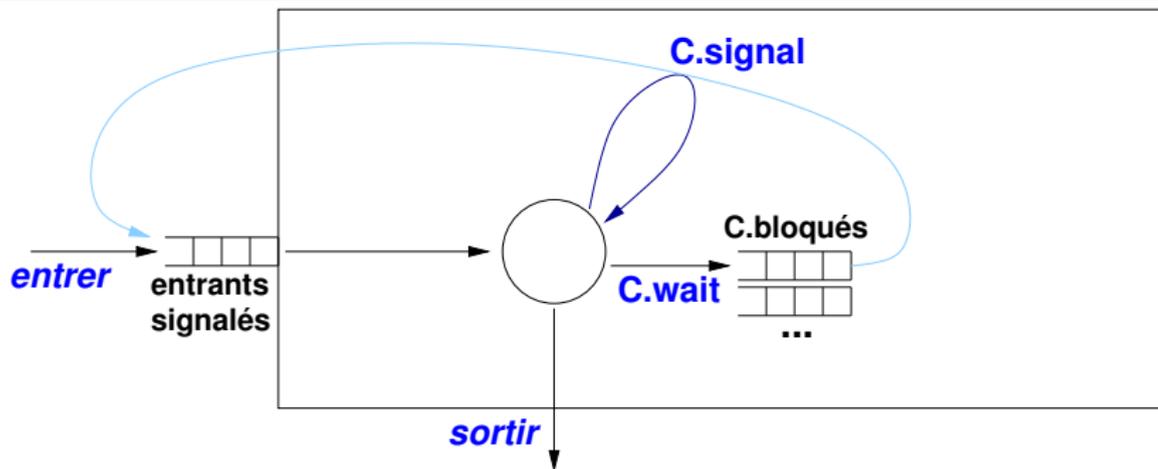
Priorité au signaleur, avec file spécifique des signalés



C.signal()

- si la file des bloqués sur *C* est non vide, 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, 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

Travail délégué avec 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 TravailDéposé.
- 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 TravailDéposé.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.

Et donc ?

- Priorité au signalé : garantit que l'activité réveillée obtient l'accès au moniteur **dans l'état où il était lors du signal**.
 - Raisonnement simplifié
 - Absence de famine facile
- Priorité au signaleur : le réveillé obtient le moniteur **ultérieurement**, éventuellement après que d'autres activités ont eu accès au moniteur.
 - Implantation du mécanisme plus simple et plus performante
 - Nécessité de **tester à nouveau** la condition de déblocage au réveil du signalé
 - Absence de famine/vivacité plus difficile



Simplifier l'expression de la synchronisation ?

Idée

Attente sur des **prédicats**, plutôt que sur des événements (variables de type condition)

→ opération : $wait(B)$ où B est une expression booléenne

Exemple : travail délégué, avec $wait(\text{prédicat})$

variables d'état : req, rés

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

`déposer_travail`(in t)

req ← t

`lire_résultat`(out r)

$wait(\text{rés} \neq \text{null})$

r ← rés

rés ← null

`prendre_travail`(out t)

$wait(\text{req} \neq \text{null})$

t ← req

req ← null

`rendre_résultat`(in y)

rés ← y

Pourquoi *wait*(prédicat) n'est-il pas disponible en pratique ?

Efficacité problématique : évaluer B à chaque nouvel état (= à **chaque** affectation), et pour **chacun** des prédicats attendus
→ gestion de l'évaluation laissée au programmeur.

- une variable condition (P_valide) est associée à chaque prédicat (P)
- $wait(P)$ est implantée par
si $\neg P$ **alors** $P_valide.wait()$ **fsi**
- le programmeur a la possibilité de signaler ($P_valide.signal()$) aux instants/états (pertinents) où P est valide

Principe

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

Plan

- 1 Spécification
 - Définition
 - Transfert du contrôle exclusif
- 2 Applications
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
 - Barrière
- 3 Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Régions critiques



Méthodologie

- 1 Déterminer l'**interface** du moniteur
- 2 Énoncer informellement les **prédicats d'acceptation** de chaque opérations
- 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 Pour chaque prédicat d'acceptation, définir une **variable condition**
- 6 **Programmer** les opérations
- 7 **Vérifier** l'invariant et les réveils



Schéma standard d'une opération

Si prédicat d'acceptation est faux alors

Attendre (*wait*) sur la variable condition associée

finsi

{ (1) État nécessaire au bon déroulement }

Maj de l'état du moniteur (action)

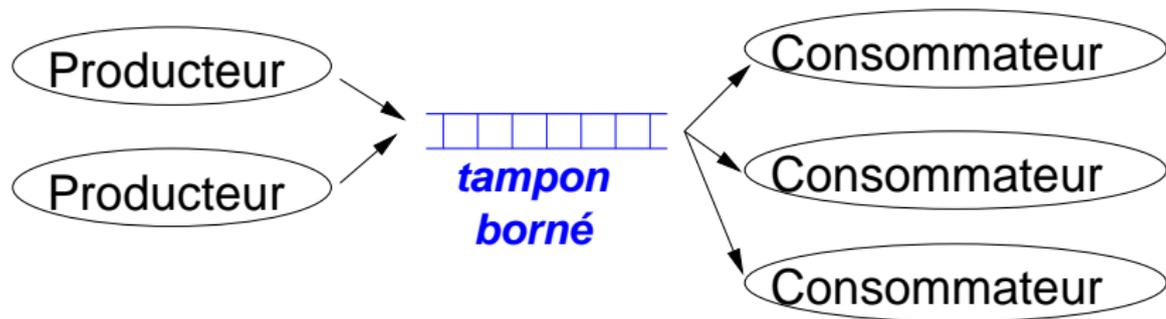
{ (2) État garanti }

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

Vérifier, pour chaque variable condition, que chaque précondition à `signal` (2) implique chaque postcondition de `wait` (1) de la variable condition correspondante.



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

- 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 : $nbOccupées < N$
 - retirer : $nbOccupées > 0$
- 4 Invariant : $0 \leq nbOccupées \leq N$
- 5 Variables conditions : PasPlein, PasVide



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 : $nbOccupées < N$
 - retirer : $nbOccupées > 0$
- 4 Invariant : $0 \leq nbOccupées \leq N$
- 5 Variables conditions : PasPlein, PasVide



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 : $nbOccupées < N$
 - retirer : $nbOccupées > 0$
- 4 Invariant : $0 \leq nbOccupées \leq N$
- 5 Variables conditions : PasPlein, PasVide



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

- 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.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
```

27

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)
```

```
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 (cf interblocage).
- 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
 - retirer(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
 - retirer(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
 - retirer(q) : rien
- 3 Variables d'état :
 - `nbDispo` : `natural`
 - `demander(p)` : `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
 - retirer(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
 - retirer(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
 - retirer(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 – 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  $\leftarrow \text{nbDispo} + q$ 
si nbDispo  $\geq$  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  $\leftarrow \text{nbDispo} + q$ 
si nbDispo  $\geq$  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  $\neg$ (nbDispo  $<$   $p$ ) alors  
    demande  $\leftarrow p$   
    AssezDeRessources.wait    -- au plus un bloqué ici  
    demande  $\leftarrow 0$   
finsi  
nbDispo  $\leftarrow$  nbDispo  $- p$   
Sas.signal    -- au suivant de demander
```

libérer(q)

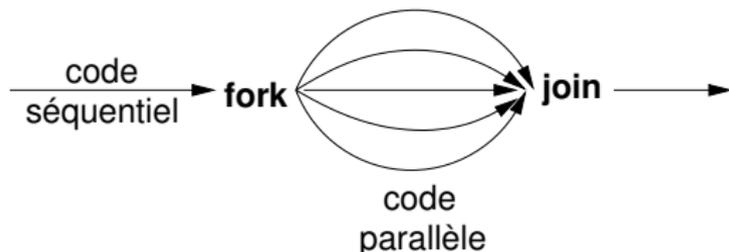
```
nbDispo  $\leftarrow$  nbDispo  $+ p$   
si nbDispo  $\geq$  demande alors  
    AssezDeRessources.signal  
finsi
```

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

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 processus 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

- 1 Interface :
 - `franchir()`
- 2 Prédicats d'acceptation :
 - `franchir()` : N processus 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

- 1 Interface :
 - `franchir()`
- 2 Prédicats d'acceptation :
 - `franchir()` : N processus 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

- 1 Interface :
 - `franchir()`
- 2 Prédicats d'acceptation :
 - `franchir()` : N processus 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

- 1 Interface :
 - `franchir()`
- 2 Prédicats d'acceptation :
 - `franchir()` : N processus 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

- 1 Interface :
 - `franchir()`
- 2 Prédicats d'acceptation :
 - `franchir()` : N processus 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 – opération

franchir()

```
nbArrivés++
si  $\neg$ (nbArrivés =  $N$ ) alors
    BarrièreLevée.wait
finsi
{ nbArrivés =  $N$  }
BarrièreLevée.signal // réveil en chaîne du suivant
nbArrivés--          // ou nbArrivés  $\leftarrow \emptyset$ 
```

Note : On pourrait remplacer le réveil en chaîne par :
si nbArrivés= N alors BarrièreLevée.broadcast

Priorité au signaleur ?

- $\geq N$ peuvent passer
- Remplacement du si en tant que : seul le premier passe

franchir(), priorité au signaleur

```
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.broadcast
finsi

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

Plan

- 1 Spécification
 - Définition
 - Transfert du contrôle exclusif
- 2 Applications
 - Méthodologie
 - Producteurs/consommateurs
 - Allocateur de ressources
 - Barrière
- 3 Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Régions critiques



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

- 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 sexe / 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)

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))  $\neq$  0  $\vee$  attente(inv(g))  $\geq$  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.

Régions critiques

- Éliminer les variables conditions et les appels explicites à `signal` \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 débloquer éventuellement.



Régions critiques

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
```

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

- 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

