

Programmation Concurrente

Systèmes Concurrents

- **cooperation:** activités *se connaissent*
- **competition :** activité *s'ignorent*

augmentation de la puissance de calcul + économies via mutualisation des donnés

-> services répartis + activités multi-processeur

=> **archi multi-proc pour améliorer la puiss de calcul**

Parallélisme: Exécution simultanée de plusieurs codes

Concurrence: Structuration d'un programme en activités \pm indépendantes qui interagissent et se coordonnent.

	Pas de concurrence	Concurrence
Pas de parallélisme	prog séquentiel	multi activités sur un mono-proc
Parallélisme	parallélisation automatique / implicite	multi activités sur un mono-proc

Activités \pm simultanées -> **explosion de l'espace d'état**

Interdépendance des activités -> **non déterministe**

Cohérence informatique/mémoire:

- **cohérence séquentielle:** résultat execution // est le même que celui d'une execution seq qui respecte l'ordre des proc
- **cohérence PRAM:** Les écritures d'un même processeur sont vues dans l'ordre où elles ont été effectuées: des écriture de processeurs différent peuvent être vues dans des ordres différents.
- **cohérence lente:** une lecture retourne une valeur précédemment écrite, sans remonter dans le temps.

Activité:

- exécution d'un programme séquentiel
- exécutable par un processeur
- entité logicielle
- interruption et commutable.

Activité communiquant par messages:

- communication par transfert de données (messages)
- coordination implicite (communication)
- designation nécessaire du destinataire (canal)

Contrôler:

- par progression et les interactions de chaque activité
 - assurer leur protection réciproque
- => **Attente par blocage suspension de l'activité**

protocole -> séquences d'actions autorisée

Décrire:

- Compter les actions/changements d'états, les relier entre eux
- Triplet de Hoare: (précondition/action/postcondition)

L'exclusion mutuelle (protocole d'isolation)

- **Section critique:** S_1, S_2 sections critiques qui doivent chacune être exécutées de manière atomique
-> Résultat concurrente S_1 et S_2 même que une des exécutions seq $S_1; S_2$ ou $S_2; S_1$
-> Contrôle l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle) ou par effets de $S_1 S_2$ (contrôle de concurrence)
- **Prop:** au plus une activité en cours d'exécution d'une section critique (sûreté)
si une demande → activité qui demande à entrer sera admise (progression)
si activité demande à entrer, elle finira par entrer (vivacité individuelle)

Implémentation: Test And Set ; Fetch And Add ; Ordonnanceur avec priorités; système de fichiers

-> Utilisation de verrous avec méthodes **acquire** et **release**

Sémaphore

Gestion des interactions entre activités (isoler, synchroniser...)

- **Sémaphore:** encapsule un entier: ≥ 0 : opération **down** (décrémente le compteur bloque signal avant de débloquent) opération **up** (incrémente le compteur)

Pour eut Sémaphore: occurrence $E := 0$ -> signalement de la présence **s.up()** attendre et consommer **s.down()**

- **Sémaphore booléen:** verrou/lock
- **Allocateur de ressources:** N ressources, 2 opérations allouer et libérer -> sémaphore avec N jetons

Interblocage

- **Allocation de ressources multiples:** gérant -> demande + libère (rend réutilisable + libère à la terminaison)
- **Correction:**
 - sûreté : rien de mauvais ne se produit (exclusion mutuelle, invariants du programme)
 - vivacité: qq ch de bon finit par se produire (équité, absence de famine, terminaison de boucle) -> p.8
- **Famine:** Une activité est en famine lorsqu'elle attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite)
- **Interblocage:** Allocation de ressources réutilisables, non réquisitionnables, non partageables, en quantité entières et finies, dont l'usage est indépendant de l'ordre d'allocation. (entrelacement peut entraîner de l'interblocage..)
- **Def:** Un ensemble d'activités est en interblocage (dead lock) \Leftrightarrow toute activité de l'ensemble est en attente d'une ressource qui ne peut être libérée que par une autre activité de l'ensemble.

Absence de famine => Absence d'interblocage

L'interblocage est un état stable.

Prévention: empêcher la formation de cycles

Détection + guérison: le détecter et l'éliminer

- *Eviter l'accès exclusif*
- *Eviter la redemande bloquante*
- *Eviter l'attente circulaire*

Moniteur

module exportant des procédures + contraintes d'exclusion mutuelle + synchro interne

- **Variable condition:** wait (bloque activité libère accès exclu au moniteur)
signal (si activités bloquées sur C elle en débloque une)

Code exécuté en exclu mutuelle

Méthodologie:

Déterminer l'interface du moniteur

Énoncer les prédicats d'acceptation de chaque opération

Déduire es variables d'états -> écrire les prédicats

Formuler l'invariant du moniteur et les prédicats d'acceptation

Pour chaque prédicat, définir une variable condition

Programmer

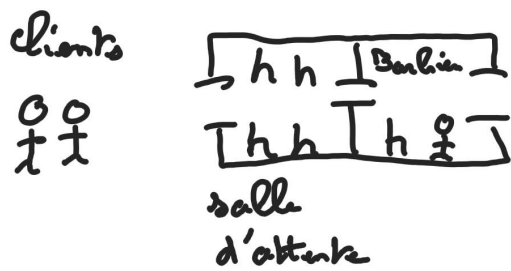
Rendez-vous **Adapté à la répartition \neq semaphore -> moniteur -> centralisé**

Comme modèle client/serveur

Tâche:

- activité
 - demander un RDV avec une autre tâche
 - attendre un RDV soit un/plusieurs point(s) d'entrée
-

1. Le problème du barbier



Code des activités

Client

```
boucle
  a) entrer dans la salle d'att
  -- lit les revues
  b) s'asseoir dans le fauteuil du barbier
  -- rêveasse pdt le rasage
  c) partir
  --- va travailler
finBoucle
```

Barbier

```
boucle
  debuter rasage
  -- raser le client
  terminer rasage
  -- va au bistrot
finBoucle
```

Interface du moniteur	Condition d'acceptation	Variables d'état	Prédicats d'acceptation
entree_SA	la salle d'att n'est pas pleine	nbPlacesLibres : nat	nbPlacesLibres > 0
SAsseoir	le fauteuil est libre	fauteuilLibre : bool	fauteuilLibre
Partir	le client n'a plus de barbe	barbePresente : bool	¬barbePresente
Débuter_rasage	il y a un client barbu dans le fauteuil		barbePresente (^ ¬fauteuilLibre)
Terminer_rasage	---		true

Invariants

```
inv 0 ≤ nbPlacesLibres ≤ N
inv barbePresente => ¬fauteuilLibre
```

Opération

```
si ¬CA alors
    VC.attendre
finSi
màj des variables d'état
déblocage
```

Variables Conditions

Salle
Fauteuil
ClientBarbu
Rasé

Codage

entrée

```
tantQue ¬(nbPlacesLibres > 0) faire
    Salle.wait
finTantQue
{précondition : nbPlacesLibres > 0}
nbPlacesLibres--
{nbPlacesLibres >= 0}
```

SAsseoir

```
tantQue ¬(fauteuilLibre) faire
    Fauteuil.wait
finTantQue
{fauteuilLibre}
fauteuilLibre <- false
nbPlacesLibres++
barbePresente <- true
{¬fauteuilLibre ^ barbePresente ^ (nbPlacesLibres > 0)}
Salle.signal
ClientBarbu.signal
```

Partir

```
tantQue barbePresente faire
    Rasé.wait
finTantQue
{¬barbePresente}
fauteuilLibre <- true
{fauteuilLibre}
Fauteuil.signal
```

DebutRasage

```
tantQue ¬barbePresente faire
    ClientBarbu.wait
finTantQue
```

TerminerRasage

```
barbePresente <- false
{barbePresente}
Rasé.signal
```

⚡ **Exécution des opérations en exclusion mutuelle**

2. Les lecteurs et les rédacteurs

Interface	Conditons d'acceptation	Variables d'état	Prédicat
Demander_Lecture DL	pas d'écriture en cours	nbLecteurs : nat (nL)	$nR = 0$
Terminer_Lecture TL	---		true
Demander_Ecriture DE	pas de lecture ni d'écriture	nbRedacteur : nat (nR)	$nL = 0 \wedge nR = 0$
Terminer_Ecriture TE	---	nbRedAtt : nat	true

Code d'une activité (bon comportements) : $((DL; TL) + (DE; TE))^*$

Invariants

inv $nR \leq 1 \wedge (nL = 0 \vee nR = 0)$

Variables Conditions

AccèsLecture
AccèsEcriture

Codage

DL

```
si ¬(nR = 0 ∧ nbRedAtt = 0) alors
  AccèsLecture.wait
finSi
nL++
{nR = 0 ∧ nL >= 1}
AccèsLecture.signal    (réveil en chaine)
```

TL

```
{nR = 0 ∧ nL >= 1}
nL--
si nL = 0 alors
  {nL = 0 ∧ nR = 0}
  AccèsEcriture.signal
finSi
```

DE

```
si ¬(nL = 0 ∧ nR = 0) alors
  nbRedAtt++
  AccèsEcriture.wait
  nbRedAtt--
finSi
{nL = 0 ∧ nR = 0}
nR++
```

TE

```
{nL = 0 ^ nR = 1}
nR--
{nL = 0 ^ nR = 0}
si nbRedAtt > 0 alors
    AccèsEcriture.signal
sinon
    AccèsLecture.signal
finSi
```

Stratégie FIFO

Méthodologie pour FIFO :

- suivre méthodologie classique
- mais une seule variable condition (FIFO)
- identifier les bugs
- bidouiller

Interface	Conditons d'acceptation	Variables d'état	Prédicat
DL	pas d'écriture en cours	nbLecteurs : nat (nL)	$nR = 0$
TL	---		true
DE	pas de lecture ni d'écriture	nbRedacteur : nat (nR)	$nL = 0 \wedge nR = 0$
TE	---	(nAtt : nat)	true

Invariants

inv $nL = 0 \vee nR = 0 ; nR \leq 1$

inv $nAtt > 0 \Rightarrow nL > 0 \vee nR > 0$

Variable Condition

Accès (FIFO)

Sas

Codage

DL

```
si  $\neg(nR = 0 \wedge \text{Accès.empty})$  alors
    AccèsLecture.wait
finSi
{nR = 0}
nL++
{nR = 0  $\wedge$  nL >= 1}
Accès.signal    (*réveil en chaine*)
```

TL

```
{nR = 0  $\wedge$  nL >= 1}
nL--
si nL = 0 alors
    {nL = 0  $\wedge$  nR = 0}
    Accès.signal
finSi
```

DE

```
si ¬(nL = 0 ^ nR = 0) alors
    Accès.wait
    si nL > 0 alors
        Sas.wait
    finSi
finSi
{nL = 0 ^ nR = 0}
nR++
```

TE

```
{nL = 0 ^ nR = 1}
nR--
{nL = 0 ^ nR = 0}
Accès.signal
finSi
```

Vérification

vérifier que chaque préconditions à `VC.signal` \Rightarrow postconditions `VC.wait`

- $2 \Rightarrow 1$: ✓
- $2 \Rightarrow 4$: ✗
- $3 \Rightarrow 1$: ✓
- $3 \Rightarrow 4$: ✓
- $5 \Rightarrow 1$: ✓

```
si Sas.empty alors
    Accès.signal
sinon
    Sas.signal
finSi
```

3. Allocateur de ressources

N ressources équivalentes à usage exclusif (ex: *page mémoire*)

Code d'un processus :

- $k \leftarrow$ nb de ressources nécessaires
- demander k ressources
- ...
- libérer k ressources

Attention

Pas 2 demandes consécutives sans libération entre.

Stratégie priorité aux petits demandeurs

Originalité : lors de libérer(k), on peut débloquer 0, 1 ou plusieurs activités.

Interface	Condition d'acceptation	Variables d'état	Prédicats d'acceptation
demander(k)	au moins k ressources libres	nbDispo : nat	$\text{nbDispo} \geq k$
libérer(k)	---		true

Invariants

inv $0 \leq \text{nbDispo} \leq k$

Variable Condition

Accès[N]

Codage

demander(k)

```
si ¬(nbDispo >= k) alors
    att[k]++
    Accès[k].wait
    att[k]--
finSi
nbDispo <- nbDispo - k
réveiller_suivant(k)  (*réveil en chaine*)
```

libérer(k)

```
nbDispo <- nbDispo + k
réveiller_suivant(nbDispo - k)
```

réveiller_suivant(départ)

```
i <- départ
tantQue Accès[i].empty ^ i <= nbDispo faire
    i++
finTantQue
si i <= nbDispo alors
    Accès[i].signal
finSi
```

Améliorations

demander : réveiller_suivant -> début à k

libérer : réveiller_suivant -> début à n

☰ Variantes de stratégie :

- Petits demandeurs -> famine des gros demandeurs
 - Priorité aux gros demandeurs -> faible parallélisme
 - Best-fit : débloquent le plus possible
 - descendre jusqu'à trouver une activité (ou personne)
 - maximise utilisation des ressources en réduisant
-

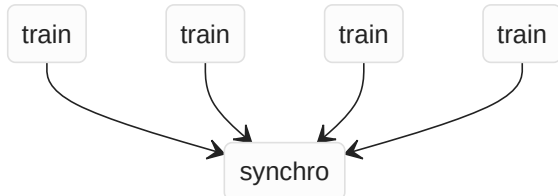
4. Problème de voie unique

Tronçon à voie unique, objectif : pas de de train en sens opposé.

Version simplifiée :

- capacité non bornée
- famine possible

Principe



Opération bloquante

- Envoyer un message sur un canal
CSP: `canal!valeur` / Go: `canal<-valeur`
- Recevoir un message depuis un canal
CSP: `canal?variable` / Go: `variable:=<-canal`

Alternative :

- action au choix
- ensemble de reception/émission

Interface : canaux ?

Interface

- entrerEO
- entrerOE
- sortir

Code d'un train

```
*[  entrerEO!_;  
    ...  
    sortie!_;  
    ...  
    entrerOE!_;  
    ...  
    sortie!_;  
]
```

où `_` est le message vide

```

for {
  entrerEO <- _
  ...
  sortie <- _
  ...
  entrerOE <- _
  ...
  sortie <- _
}

```

Construction de l'activité de synchronisation : approche par conditions

- Énoncer les conditions d'acceptation par canal
 entrerEO : pas de train en sens OE
 entrerOE : pas de train en sens EO
 sortie : toujours faisable

Variable d'état (interne à l'activité de synchro)

```

nbEO
nbOE

```

Invariant

inv (nbOE = 0) v (nbEO = 0)

Activité de synchro

```

boucle
  alternative
    quels sont les canaux ouverts selon l'état courant
    selon la réception -> nouvel état
finboucle

```

```

*[
  nbOE = 0 -> entrerEO?_; nbEO++
  []
  nbEO = 0 -> entrerOE?_; nbOE++
  []
  sortir?_; nbEO > 0 -> nbEO--
  [] _ -> nbOE--
]

```

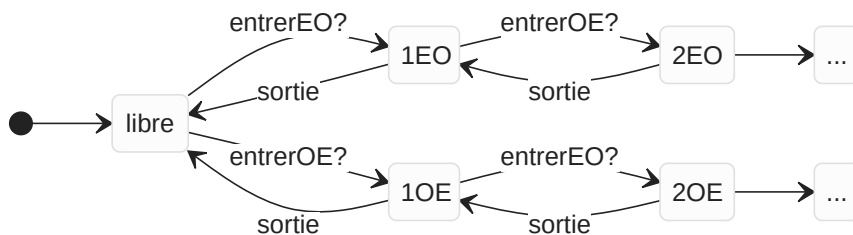
```

for {
  select {
    case <- when(nbEO = 0, entrerOE):
      nbOE++
    case <- when(nbOE = 0, entrerEO):
      nbEO++
    case <- sortie:
      if nbEO > 0 {nbEO--} else {nbOE--}
  }
}

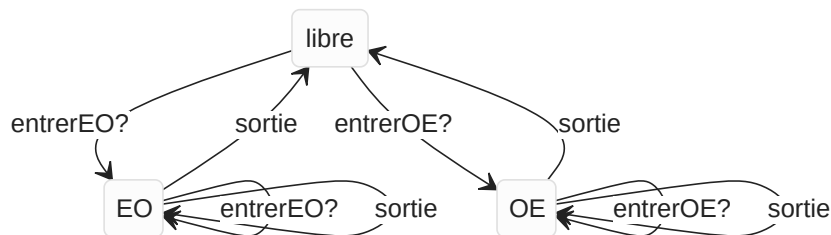
```

Approche par automate

L'état = l'ensemble de canaux ouverts (sur lesquels un message peut-être reçu)



Automate généralisé



Code

```
etat := libre
for {
  if etat == libre {
    select {
      case <- entrerEO:
        etat := EO
        nb := 1
      case <- entrerOE:
        etat := OE
        nb := 1
    }
  } else if etat == EO {
    select {
      case <- entrerEO: //when(nb < N, entrerEO)
        nb++
      case <- sortir:
        nb--
        if nb == 0 {etat := libre}
    }
  } else {
    //etat == OE
    //sym de EO
  }
}
```

Capacité borné à N

- Approche condition
entrerEO : $(nbOE = 0) \wedge (nbEO < N)$
entrerOE : $(nbEO = 0) \wedge (nbOE < N)$
- Approche automate
conditionner certaines ouverture

Famine

Un flux continu de trains dans un sens, tel qu'il y a toujours au moins un train sur la voie unique.

Empêche de manière permanente l'entrée des trains en sens opposé.

-> ne pas laisser entrer trop de trains successifs dans un sens s'il y a des demandes dans l'autre sens.

- soit savoir s'il y a des demandes d'écriture bloquées sur un canal => NON (pas fourni)
- soit enrichir l'interface

```
boucle
    preparer_entrerE0!_
    ...
    entrerE0!_
    ...
    sortie!_
    ...
finboucle
```

5. Tournoi de Bridge


Salle avec des tables (en nombre quelconque) de capacité de 4 chacune.

Il faut maintenir toutes les tables complètes (ou vide)

Les joueurs peuvent entrer/sortir

☰ On peut accepter :

- un échange
- 4 entrées
- 4 sorties

 Interface

entrer , sortir

préparer_entrée , préparer_sortie

Code d'un joueur

```
boucle
    preparer_entree!_
    entrer!_
    //joue
    preparer_sortie!_
    sortir!_
finboucle
```

Approche conditions

 Variables d'état

nbdemE

nbdemS

(nb dans la salle est inutile, sauf si nb de table est borné)

Code

```
*[
    préparer_entrée?_; ne++;
    []
    préparer_sortie?_; ns--;
    []
    ne >= 1 || ns >= 1 -> entrer?_; sortir?_; ne--; ns--;
    []
    ne >= 4 -> entrer?_;entrer?_;entrer?_;entrer?_;
    ne = ne - 4;
    []
    ns >= 4 -> sortir?_;sortir?_;sortir?_;sortir?_;
    ns = ns - 4
]
```

```
for {
    select {
        case <- préparer_entrer:
            ne++
        case <- préparer_sortie:
            ns++
        case <- when(ne >= 1 && ns >= 1, entrer):
            <- sortir
            ne--; ns--;
        case <- when(ne >= 4, entrer):
            <- entrer; <- entrer; <- entrer;
            ne = ne - 4;
        case <- when(ns >= 4, sortir):
            <- sortir; <- sortir; <- sortir;
            ns = ns - 4;
    }
}
```

Approche automate

état = (ne,ns)

