

Travaux Dirigés

Introduction

Ce document présente une série de problèmes de synchronisation. Ces problèmes sont exposés de manière à pouvoir être traités dans divers contextes d'exécution (mémoire partagée, processus communiquants), et avec les diverses primitives proposées par les langages et modèles de synchronisation (sémaphores, rendez-vous, moniteurs, expressions de chemins...).

Une application réalisée par plusieurs activités concurrentes comportera en général

- la description du code exécuté par chacune des activités;
- la description des variables globales objets partagés par les activités;
- un programme principal, qui sera lui même une activité, à partir de laquelle seront lancées, implicitement ou explicitement les activités constituant l'application proprement dite.

Par exemple, une application réalisant un schéma d'interaction producteur/consumateur au moyen d'un tampon de taille bornée N (et utilisant des sémaphores pour coordonner les activités) pourra avoir la structure suivante :

```

programme Prod_Conso;
    import lance_activité, sémaphore, tampon, ligne, créer, utiliser;
    const C = 1; {nombre de consommateurs}
    P = -1; {nombre de producteurs}
    N = -1; {taille du tampon}
    var t : tampon[N] <ligne>;
        mutex conso_produ; vide, plein : sémaphore;
        procedure producteur;
            far lgn : ligne;
            début (producteur)
                cycle
                    créer(lgn);
                    vide();
                    mutex_prod.P();
                    mutex_prod.V();
                    fin
            fin producteur;
        procédure consommateur;
            début (consommateur)
                plein.P();
                mutex_cons.P();
                lire(tampon[lgn]);
                mutex_cons.V();
                vide.V();
                utiliser(lgn);
            fin consommateur;
        fin consommateur;
    début (prod_cons)
        mutex_Proc.init();
        utiliser(N);
        plein.init();
        mutex_cons.init();
        pour i := 1 à C faire lancer activité(consommateur); {lance C consommateurs}
        pour i := 1 à P faire lancer activité(producteur); {lance P producteurs}
    fin Prod_Cons.

```

Problèmes de synchronisation

1 Allocateur de ressources

Un système comporte souvent des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les imprimantes. Lorsqu'un processus utilise une imprimante, aucun autre processus ne peut et ne doit y accéder. Par ailleurs, il est courant qu'un même service puisse être assuré de manière équivalente par n'importe laquelle des ressources d'un pool de ressources. Ainsi, un utilisateur peut avoir le choix entre plusieurs imprimantes, ce qui lui évite d'avoir à attendre la disponibilité d'une imprimante occupée, ou éteinte, ou à court de fournitures.

On envisage un protocole permettant à un processus d'acquérir par une seule action plusieurs ressources (d'une même catégorie). Dans la mesure où l'on ne souhaite pas programmer effectivement l'allocation des ressources, mais où l'on se concentre sur la coordination entre demandeurs et libérateurs de ressources, on ne considère qu'une catégorie de ressources, et on ne manipulera pas les références aux ressources proprement dites. On considérera donc que les actions assurant le contrôle d'accès à de telles ressources sont les suivantes :

— Allouer (nbDemandé : entier);

— Libérer (nbRendu : entier);

Programmer les algorithmes de ces opérations, en envisageant différentes politiques d'allocation :

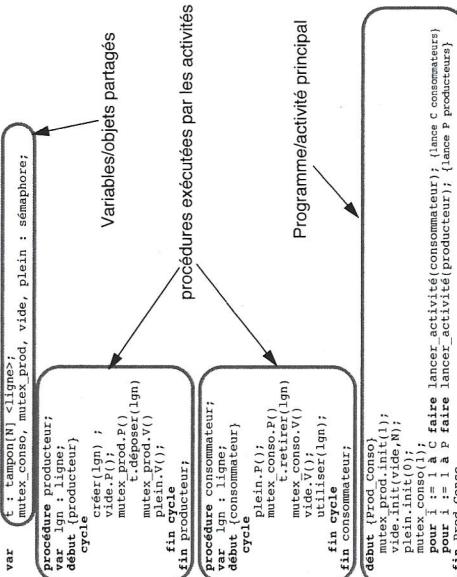
ordre FIFO : les demandes sont servies dans l'ordre chronologique d'arrivée.

Cette stratégie permet de garantir l'absence de famine sur toute demande.

priorité aux petits demandeurs : lorsque des ressources sont libérées, la demande portant sur le plus petit nombre de ressources est servie en priorité.

Cette stratégie permet de servir un maximum de demandes par unité de temps.

stratégie de l'ascenseur et ses variantes : les demandes sont rangées dans des files d'attentes associées aux nombres de ressources demandées (un niveau file par valeur possible). Les demandes sont servies en parcourant les différents niveaux et en servant chaque niveau selon une politique fixée (par exemple : une seule demande par niveau, à chaque passage). Cette politique permet de combiner une forme de priorité, et une forme d'équité.



2 Les philosophes et les spaghettis (Dijkstra)

Cinq philosophes, réunis pour philosopher, ont, au moment du repas, un problème pratique à résoudre. En effet, le repas est composé de spaghettis qui, selon le savoir-vivre de ces philosophes, se mangent avec deux fourchettes. Or, la table n'est dressée qu'avec une seule fourchette par couvert. Après réflexion et débat, les philosophes conviennent de suivre le protocole suivant :

1. A tout instant, chaque philosophe se trouve dans l'un des trois états suivants :
 - ou bien il mange (pendant un temps fini),
 - ou bien il attend la disponibilité de ses fourchettes,
 - ou bien il pense (pendant un temps fini) et il a alors la politesse de ne garder aucune fourchette.
2. Initialement, tous les philosophes pensent.
3. Pour manger, chaque philosophe dispose d'une place à table, qui lui est allouée exclusivement.
4. Tout philosophe qui mange, utilise sa fourchette de droite et celle de gauche. Il ne peut en atteindre d'autres. Deux philosophes voisins ne peuvent donc manger en même temps.

Chaque philosophe peut être représenté par un processus cyclique dont le code est le suivant :

```
processus Philosophe
maPlace : Place;
répéter
  PrendrePlace(maPlace);
  penser;
  PrendreFourchettes (maPlace) ;
  manger;
  ReposerFourchettes (maPlace) ;
  sans fin ;
fin Philosophe;
```

- Programmer les fonctionnalités et le protocole associés aux actions `PrendrePlace`, `PrendreFourchettes`, `ReposerFourchettes`. L'action `PrendrePlace` est appelée lorsqu'un nouveau philosophe est lancé. Elle permet en particulier d'initialiser les variables locales au processus, mais peut aussi comporter d'autres actions, selon la stratégie envisagée.
- Rechercher une stratégie permettant d'assurer simplement l'allocation exclusive des fourchettes aux philosophes, puis une stratégie optimale c'est-à-dire autorisant le maximum de parallélisme et par conséquent réalisant une utilisation optimale des ressources.
- Etudiez les possibilités de faim et/ou d'interblocage des solutions envisagées.
- Rechercher, enfin, une stratégie assurant l'absence de famine.

3 Les fumeurs de cigarettes (Patil)

Trois (gros) fumeurs, représentés par des processus *F1*, *FF* et *FT*, roulent leurs cigarettes eux-mêmes. Ils ont besoin de tabac, de feuilles, et d'allumettes. Chacun d'eux dispose en permanence d'un seul des trois ingrédients : le tabac pour *FT*, les feuilles pour *FF*, les allumettes pour *F1*. Un quatrième processus, l'agent, dispose en permanence des trois ingrédients. L'agent place sur une table deux des trois ingrédients, à sa convenance. Les fumeurs ne peuvent prendre les ingrédients placés sur la table qu'après avoir été commandés par l'agent. Les fumeurs doivent donc se coordonner afin que ce soit seulement le fumeur auquel manquent précisément les deux ingrédients fournis par l'agent qui se procure ces ingrédients. L'activité des fumeurs est cyclique. Un cycle consiste à se procurer les deux ingrédients manquants, à rouler une cigarette, à fumer la cigarette, et à rendre à l'agent les ingrédients empruntés. L'agent ne place une nouvelle paire d'ingrédients que lorsque les ingrédients qu'il a fournis auparavant lui ont été rendus.

Coordonner l'activité de l'agent et des trois fumeurs.

4 Le barbier (d'après Dijkstra)

Il s'agit de modéliser l'activité coordonnée d'un barbier et de ses clients, à l'aide de processus synchronisés.

1. Pour commencer, le barbier dispose d'un unique siège, et travaille sans local.
 - Le siège ne peut accueillir qu'un client à la fois : lorsqu'un client se présente et trouve le siège libre, il peut s'installer, pour se faire raser ; sinon il attend son tour. Un client installe sur le siège ne le quitte qu'une fois rase. Le client suivant peut alors s'installer, pour se faire raser.
2. La boutique du barbier comporte maintenant un salon, avec un unique siège, et une salle d'attente, avec N chaises. Lorsqu'un client se présente, il entre dans la salle d'attente, si l'y a des chaises libres, sinon il attend dehors qu'une chaise se libère. Les clients entrent dans le salon lorsqu le siège est libre. Un client installé sur le siège ne le quitte qu'une fois rase. Le client suivant peut alors entrer dans le salon et s'installer, pour se faire raser.
3. Variante du scénario 2) : dans le cas où un client se présente et trouve toutes les chaises de la salle d'attente occupées, il va chez un autre barbier.
4. La boutique est maintenant prospère : si la salle d'attente comporte toujours N chaises, et S sièges ont été installés dans le salon, où B barbiers officient. Quand un client se présente, il entre dans la salle d'attente, s'il y a des chaises libres, sinon il va chez un autre barbier. Lorsqu'un siège est libre (et que son tour est venu), il quitte la salle d'attente, s'installe dans le siège et attend qu'un barbier vienne le raser. Lorsqu'un barbier a fini de raser un client, il rejoint les barbiers inemployés, qui attendent qu'un client s'installe dans un siège libre.

5 Voie unique (Brinch Hansen)

Deux villes E et O sont reliées par une ligne de chemin de fer qui comprend un tronçon à voie unique.



Pour s'engager sur le tronçon à voie unique les processus représentant les trains devront faire appel aux actions suivantes :

Entrer(direction)

Sortir()

avec : **type** direction = (OE, NS)

Les trains font la navette entre E et O. Leur comportement peut donc être simulé par le type de processus suivant :

processus train

maDirection : direction ;

Démarrer(maDirection); -- fixe la direction initiale

répéter

arriver_au_tronçon();

Entrer(maDirection) ;

passer_le_tronçon;

Sortir() ;

arriver_à_destination();

maDirection := successeur(maDirection);

sans fin ;

fin train;

Ecrire les algorithmes des actions Entrer et Sortir lorsque :

1. un train seulement peut se trouver sur le tronçon à voie unique ;
2. un nombre illimité de trains peuvent circuler en même temps sur le tronçon à voie unique (à condition qu'ils aillent dans le même sens) ;
3. au plus N trains peuvent circuler en même temps sur le tronçon à voie unique. Considérer les risques de privation dans ce cas, et donner les moyens d'éviter celle-ci.

7 Un tournoi de bridge

On désire modéliser le déroulement d'un tournoi de bridge. L'organisation du tournoi impose d'assurer que le nombre de personnes présentes dans la salle du tournoi soit toujours un multiple de 4. Il faut donc contrôler les entrées et les sorties selon les contraintes suivantes :

- Un joueur peut entrer si un autre est prêt à sortir (échange) ou s'il y a déjà

- 3 autres joueurs en attente d'entrée (le groupe de 4 peut alors entrer).

- Un joueur peut sortir si un autre est prêt à entrer (échange) ou s'il y a déjà

- 3 autres joueurs en attente de sortie (le groupe de 4 peut alors sortir).

Remarque : dans le cas de la sortie de 4 joueurs appartenant à des tables différentes, on ne s'intéresse pas à la réorganisation des tables de jeu pour continuer le tournoi.

L'accès à la salle est contrôlé par l'appel des actions Entrer et Sortir. Le comportement d'un joueur est modélisé par le type de processus suivant :

processus Joueur

{se rendre sur le lieu du tournoi()}

Entrer() ;

{Joueur()}

Sortir() ;

{rentrer chez soi()}

fin joueur ;

fin joueur ;

Questions

1. On définit 2 variables d'état *ne* et *ns* variant dans l'intervalle [0..4]. Lorsqu'aucune opération Entrer ou Sortir n'est en cours :

- La variable *ne* compte le nombre de entrants en attente;

- La variable *ns* compte le nombre de sortants en attente.

A l'aide de ces variables exprimez les prédicts suivants :

- (a) La condition pour qu'une opération Entrer provoque un échange avec un sortant.

- (b) La condition pour qu'une opération Sortir provoque un échange avec un entrant.

- (c) La condition de blocage d'un entrant.

- (d) La condition de blocage d'un sortant.

- (e) L'invariant global liant les variables d'état *ne* et *ns*.

2. Déduire des spécifications précédentes, les conditions de synchronisation nécessaires.

3. En déduire les algorithmes des opérations Entrer et Sortir. Annotez les algorithmes de manière à prouver leur correction.

6 Un album sociofinancé

On considère un album réalisé par un musicien pionnier (mais quelque peu intéressé) : l'album n'est produit que si une somme d'argent suffisante est réunie par le public. Pour cela, chaque personne peut offrir une somme arbitraire, et quand la somme exigée par le musicien est atteinte, les personnes ayant financé peuvent écouter l'album.

Ecrire l'algorithme des actions assurant ce protocole de financement participatif :

- payer(somme : natural),

- écouter()

8 Les lecteurs et les rédacteurs (Courtosis)

On considère 2 classes de processus, des lecteurs et des rédacteurs, accédant à un fichier partagé.

Ce fichier peut être lu simultanément par plusieurs lecteurs, mais un seul rédacteur peut écrire à un instant donné, à condition qu'aucun lecteur ne lise (exclusion mutuelle entre lecteurs et rédacteurs).

Les actions assurant le protocole d'accès au fichier partagé sont les suivantes :

DébuterLecture,

TerminerEcriture,

Programmer la synchronisation des lecteurs et rédacteurs dans les cas suivants :

Les lecteurs sont prioritaires sur les rédacteurs : un rédacteur ne peut accéder au fichier que si aucun lecteur n'est en cours ou en attente de lecture. Un rédacteur ne peut cependant pas être interrompu par un lecteur.

Lecteurs coalisés : si un rédacteur possède l'accès au fichier, lecteurs et rédacteurs sont mis en attente, puis servis dans l'ordre chronologique de leur requête d'accès. Cependant, dès qu'un lecteur obtient l'accès au fichier, il libère l'ensemble des lecteurs en attente. Enfin, lorsqu'un lecteur possède l'accès au fichier, seuls les rédacteurs sont mis en attente.

Lecteurs coalisés, avec équité. La solution précédente n'évite pas la possibilité d'une fanuie pour les rédacteurs.

- Pourquoi ?
- Amender la politique précédente, afin d'assurer une forme d'équité dans l'accès au fichier (tout processus demandant un accès finit par l'obtenir).
- Corriger le protocole en conséquence.

Accès dans l'ordre chronologique d'arrivée (ordre FIFO) sans considérer la classe (lecteur ou rédacteur) des processus.

Les rédacteurs sont prioritaires sur les lecteurs Il s'agit du cas symétrique au premier cas. Un lecteur ne peut lire que si aucun rédacteur n'est en cours d'écriture ou en attente d'accès.

9 Gestion d'un carrefour

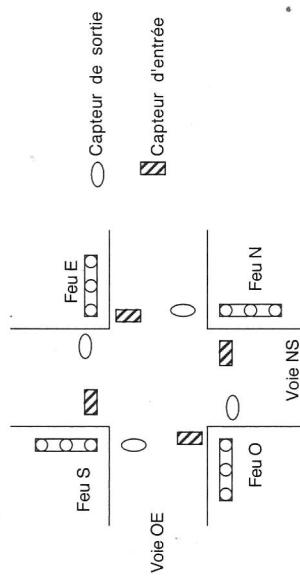
On désire assurer la commande des feux tricolores d'un carrefour par ordinateur. On suppose pour cela que des capteurs placés sous les voies de circulation détectent le passage des véhicules en entrée et en sortie.

Pour traverser (régulièrement !) le carrefour, les processus devront faire appel aux actions suivantes :

Entrer(voie)

Quitter()

avec : **type** voie = (OE, NS)



Chaque fois qu'un véhicule entre, l'une des 2 actions Entrer(OE) ou Entrer(NS) est appellée. De même, chaque fois qu'un véhicule sort, l'action Quitter() est appellée. Plusieurs véhicules peuvent se trouver dans le carrefour.

1. Expliquez pourquoi l'origine du véhicule (voie et direction exactes d'entrée) n'est pas une information nécessaire pour l'opération Quitter().

2. On définit les règles suivantes de commande des feux :

- Les véhicules arrivant sur la voie ouverte peuvent passer tant qu'il n'y a aucun véhicule en attente sur l'autre voie. Cependant, s'il existe des véhicules en attente sur l'autre voie, on limite le flot de véhicules passés successivement sur la voie ouverte à une valeur maximale N.
- Il y aura commutation de la voie ouverte chaque fois que le carrefour redévie vide, qu'il y ait ou non des véhicules en attente d'entrée.
- Lorsque le carrefour est vide et qu'aucun véhicule n'est arrêté, le feu reste dans une position fixe autorisant le passage de futurs arrivants sur l'une des 2 voies. Lorsqu'un premier véhicule arrive sur la voie ouverte, les feux restent inchangés. Mais, si ce premier véhicule arrive sur la voie fermée, les feux doivent commuter pour laisser passer ce véhicule.

Attention : Une fois qu'un flot de N véhicules est entré, la voie ouverte est fermée mais on prendra soin de n'ouvrir la voie opposée que lorsque le

carrefour sera vide. Pendant un certain laps de temps, tous les feux sont alors au rouge (les 2 voies sont fermées). On suppose que les automobilistes jouent le jeu, c'est-à-dire qu'ils ne grillent pas les feux.

- (a) Déterminer les variables d'état nécessaires au problème.

- (b) Définir les prédicts associés aux entrées.

- (c) Programmer les actions Entrer(voie) et Quitter(). On supposera disponibles les procédures permettant d'ouvrir ou fermer une voie par le passage au vert ou au rouge des feux associés à cette voie.

procédure Ouvrir(v : voie);

procédure Fermer(v : voie);

- (d) Si un automobiliste grille un feu rouge, quelle anomalie provoque-t-il dans le comportement du système ? Modifier les actions pour qu'elles résistent aux feux grillés.

Programmation concurrente - TD

Barbier

1/ Barbier

```
while (true) {
    Pré : client assis
    raser
    Post : barbier terminé
    balayer
}
```

Client

```
while (true) {
    s'asseoir
    Post : client assi
    Pré : Barbier terminé
    se lever
    vivre sa vie
}
```

Spécification

Barbier ne rase pas dans le vide
 Client ne se lève pas avant rasage
 Client assis → Barbier rase.

Sémaphore : compteur entier ≥ 0

2 opérations : - up : incrémentation \rightarrow si 0 bloqué.
 down : décrémentation

Nb de sémaphores : compter le nbre de paires de pré/post.

Par convention : Etat et pas action pour les sémaphores

⚠ interblocage : client attend que le barbier fasse un truc et inversement.

2/ Client Assis : 0 Barbier Terminé : 0 Siège Libre : 1

Barbier

```
while (true) {
    ClientAssis.down
    raser
    BarbierTerminé.up
    balayer
}
```

Pré \rightarrow down
 Post \rightarrow up

Client

```
while(true) {
    SiègeLibre.down
    s'asseoir
    ClientAssis.op
    BarbierTerminé.down
    se lever
    SiègeLibre.up.
```

Un seul client assis à la fois.

Salle Attente : n

3/ Client Assis : 0 Barbier Terminé : 0 Siège Libre : 1 Salle Non Pleine : N/N+1

Barbier

Client Cr.

while (true) {
 Pré: SalleNonPleine
 Salle Non Pleine .down
 entrer
 Pré: SiègeLibre
 SiègeLibre.down
 S'asseoir sur le siège
 Post Client Assis
 Pré Barbier terminé
 Client Assis .up
 BarbierTerminé down
 selever
 Post SiègeLibre
 SiègeLibre .up
 sortir.
 Post Salle NonPleine
 Salle NonPleine .up
 Vivre sauvé .}

Au plus N clients dans la salle d'attente

Motex: 1

Client

Post d'attente bloquante

while (true) {
 Motex.down
 if (nbLibres > 0) {
 nbLibres -- ;
 Motex.up
 Pré siège Libre ...
        ~~~  
        ~~~  
 Vivre sauvé .}

Les philosophes et les spaghetti

Idee de base : 1 semaphore par fourchette

Sem Fourchettes [N] : 1.

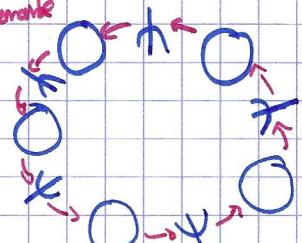
Prendre Fourchettes (maPlace)

maPlace.fourchetteGauche.down
maPlace.fourchetteDroite.down

Reposer Fourchettes (maPlace)

maPlace.fourchetteGauche.up
maPlace.fourchetteDroite.up

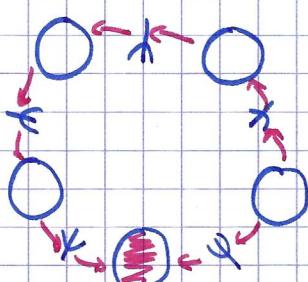
$O \rightarrow \Psi$ passe le
 $O \leftarrow \Psi$ devant



So pour éviter intransitivité :

- ① - Ψ change de côté
- ② - relâcher si attente
- ③ - Sémaphore global
- ④ - Sémaphore sur nbre philo
- ⑤ - Ordre total sur fourchettes

①



Solution corredre mais pas optimale
- processus peuvent être relâchés 1 par 1.
- cas où 1 seul ne mange

② Risque de LiveLock (On demande tt le temps au même ress mais on ne les a jamais).

③ Sem : 1 Prendre Fourchettes (maPlace)

beep; résultat false

sem.down()

résultat = (maPlace.fourchetteGauche == Libre & maPlace.fourchetteDroite == Libre)

if (résultat)

fourchetteGauche := Libre

fourchetteDroite := Libre

sem.up()

Solution correcte mais

peu concorrente.

Reposer Fourchettes (maPlace)

sem.down()

fourchetteGauche := Libre

fourchetteDroite := Libre

sem.up()

Programmation concurrente - TD3 - Moniteur.

→ Moniteur = objet dont les méthodes sont en exclusion mutuelle

* variables conditions: - attendre
- réveiller ceux qui attendent.

► Barbier : ① Interface

- Débuter Rasage
- Terminer Rasage
- S'asseoir
- Se lever

② Prédicats d'accès

- a) S'asseoir : Siège est Libre
- b) Se lever : Rasage Terminé
- c) Débuter Rasage : Client Assis Barbu
- d) Terminer Rasage : Client Assis Barbu

③ Variables d'état

- Siège Libre : booleen
- Barbu : booleen (variable qui passe à True que si la personne est assise)

④ Invariant + prédict

- Barbu \Rightarrow \neg SiegeLibre
- a) SiegeLibre
- b) \neg Barbu
- c) \neg SiegeLibre \wedge Barbu

⑦ Vérification

$$\text{Pré(signal)} \Rightarrow \text{Post(wait)}$$

⑤ Variables Conditions

- Siege Accessible
- Rasage Fini
- Client Présent

⑥ Programmer !!!

→ S'asseoir () {
| if (\neg SiegeLibre) {
| | SiegeAccessible.wait
}| }
SiegeLibre \leftarrow false
Barbu \leftarrow true
ClientAssisBarbu.signal()
}

→ Se lever () {
| if (Barbu) {
| | RasageTerminé.wait
}| }
SiegeLibre \leftarrow true
SiegeAccessible.signal
}

→ Débuter Rasage () {
| if (\neg Barbu) {
| | ClientAssisBarbu.wait
}| }
BarbePrésente \leftarrow false
RasageFinis.signal
}

→ TerminerRasage () {
| BarbePrésente \leftarrow false
| RasageFinis.signal
| }

► Les Lecteurs et Pas réducteurs Lecteurs prioritaires

① Interface

- DébuterLecture
- TerminerLecture

- DébuterEcriture
- TerminerEcriture

② Prédicats d'acquisition

- a) DébuterLecture : Pas d'écriture en cours
- b) TerminerLecture
- c) DébuterEcriture : Pas de lecture et Pas d'écriture et pas de lecteur en attente
- d) TerminerEcriture

③ Variables d'état

- EcritureEnCours : bool
- Lecteurs : int
- LecteursAttente : int

④ Invariant + Prédicat

- $\neg (\text{EcritureEnCours} \wedge \text{lecteurs} > 0)$
- a) $\neg \text{EcritureEnCours}$
- b) $\neg \text{EcritureEnCours} \wedge \text{lecteurs} == 0 \wedge \text{lecteursAttente} == 0$

⑤ Variables Conditions

- PasEcriture
- EcriturePossible

⑥ Programmation

→ DébuterLecture () {
 | if (EcritureEnCours) {
 | | LecteursAttente ++
 | | PasEcriture . wait
 | | LecteursAttente --
 | }
 | Lecteurs ++
}
}

→ DébuterEcriture () {
 | if (7 b) {
 | | EcriturePossible . wait
 | }
 | EcritureEnCours ← true
}
}

→ TerminerLecture () {
 | Lecteurs --
 | if (lecteurs == 0 \wedge lecteursAttente == 0) {
 | | EcriturePossible . signal
 | }
}
}

→ TerminerEcriture () {
 | EcritureEnCours ← false
 | if (lecteursAttente == 0) {
 | | EcriturePossible . signal
 | }
 | PasEcriture . signal All
}
}

⑦ Vérification

Allocateur de ressources

① Allocateur : N ressources
 - Allouer (nb)
 - Libérer (nb)

② Prédicats d'acceptation :

- ① Allouer : ressources disponibles
- ② Libérer

③ Variables d'états

- nbDispo : int
 nb Demandes [N, 1] : int

④ Invariant + Predicat

$$\begin{aligned} \text{nbDispo} &\geq 0 \\ \text{Allouer} : \text{nb} &\leq \text{nbDispo} \end{aligned}$$

⑤ Var Cond

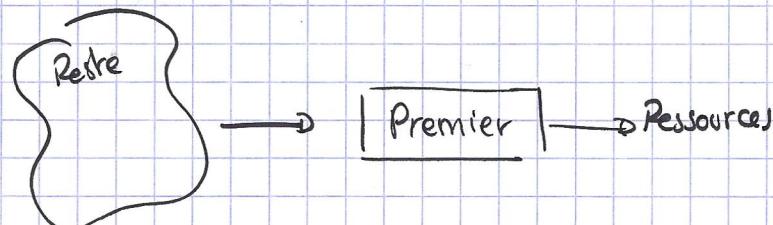
Assez Ressources [N+1]

⑥ Programmer

Allouer (nb)
 if (nb > nbDispo) {
 nb Demandes [nb] ++
 Assez Ressources [nb].wait
 nb Demandes [nb] --
 }

nbDispo -= nb
 min Demandeurs < plus petite demande
 if (min Demandeurs < nbDispo) {
 Assez Ressources [min Demandeurs].signal.
 }.

Libérer (nb)
 nbDispo += nb
 min Demandeurs < plus petite demande
 if (min Demandeurs < nbDispo) {
 Assez Ressources [min Demandeurs].signal.
 }



Allocateurs : idem

Pred : idem

Var Etat : nbDispo : int demande : int

Inv : idem.

Var Cond : Premier Reste

Allouer (nb)
 if (demande != 0) {
 Reste.wait
 }

if (nb > nbDispo) {

demande -= nb

Premier.wait

demande -= 0

}

nbDispo -= nb.

Reste.signal.

Libérer (nb)

nbDispo += nb

if (demande < nbDispo) {

Premier.signal.

}

S; Var Cond st FIFO



Philosophes et spaghetti

N Philo

Prendre fourchette (nb)

Poser fourchette (nh)

Pred Acc:

Prendre Fourchette Voisins se mangent peu.
Libérer fourchette.

Var Etat:

etat Philo [N]

Inv + Pred:

Prend Fourchette, etat Philo (no - 1) ≠ mange et etat Philo (no + 1) ≠ mange.
Libérer Fourchette.

Var Cond

Peut manger [N].

Algo:

Prendre Fourchette,

if (cond Prod Fch (Nb)) { .Etat Philo (nb) ← demandeur.

Peut manger (nb). vrai.

.Etat Philo [no] ← mange.

Libérer

Libérer Fourchette,

if (Etat Philo (no - 1) = demandeur et Etat Philo (no + 2) = mange),

Etat Philo (no - 1) ← mange

Peut manger [no - 1] si vrai.

Rendez-vous

- + Sémaphore et Moniteurs : mémoire partagée (m machine)
- + Rendez-vous : système répartis (plusieurs machines) (\rightarrow Client/Serveur)

Voie unique (Brinch Hansen)

EntrerVersEst : Personne n'utilise vers Ouest. EntrerVersOuest : Personne n'utilise vers Est. \rightarrow Prédicats d'acceptation.

Sorhir : -

EstOuest : bool true si aucun
OuestEst : bool idem
nbTransitions : int

EntrerEst
EntrerOuest
Sorhir \rightarrow Canaux

loop {
select {
when (\neg EstOuest) EntrerVersEst \leftarrow 1
nbTransitions \leftarrow 1
OuestEst = true
}}

when (\neg OuestEst) EntrerVersOuest \leftarrow 1
nbTransitions \leftarrow 1
EstOuest = true

when (EstOuest || OuestEst) Sorhir \leftarrow 1
nbTransitions \leftarrow 1
if (nbTransitions == 0) {
EstOuest = false
OuestEst = false
}

} .

Un tournoi de bridge

Débuter Entrée :

Finir Entrée : Qu'un veut sortir ou 3 autres veulent entrer

Débuter Sortie :

Finir Sortie : Qu'un veut entrer ou 3 autres veulent sortir

nb Entrants : int

nb Sortants : int

Débuter Entrée

Finir Entrée

Débuter Sortie

Finir Sortie

} → Canaux

loop {

 select {

 when (true) ← Débuter Entrée {
 nb Entrants += 1

 }

 when (true) ← Débuter Sortie {
 nb Sortants += 1

 }

 when ((nb Sortants > 0) & (nb Entrants > 0)) ← Finir Entrée ← Finir Sortie.
 nb Entrants --
 nb Sortants --

 }

 when (nb Sortants > 3) for i=1 to i=4
 ← Finir Sortie.
 nb Sortants -= 4

 }

 when (nb Entrants > 3) for i=1 to i=3
 ← Finir Entrée
 nb Entrants -= 3

 }

}

