

2^{ème} partie en parallèle : *LST Info&Miage*

Programmation système et
réseau du point de vue « Multi-
processus »

Chapitre 1 : Introduction à la Concurrency
entre processus & Exclusion Mutuelle

Chapitre 2 : **Coopération** entre processus &
Synchronisation + Communication

Coopération entre processus & Synchronisation + Communication

1. Introduction
2. Rendez-vous de N entités
3. Producteur(s) / Consommateur(s)
4. Exemple d'un tube Unix : synchronisation et communication (objet d'un chapitre à part)
5. Généralisation Exclusion Mutuelle avec des processus de type Lecteur ou Rédacteur

1 Introduction

Présentation du Problème

- Interactions inévitables entre entités
 - Volontaire : résolution collective d'un problème
 - Involontaire : utilisation concurrente de ressources
- En l'absence d'interaction
 - Exécution dans n'importe quel ordre
 - donc, possibilité de (pseudo) parallélisme
- En présence d'interactions
 - Nécessité de **synchroniser**
 - Influencer sur l'ordre dans lequel elles s'exécutent
 - L'exclusion mutuelle est une forme de synchronisation involontaire, qui *sérialise* l'exécution des Sections Critiques: force l'exécution en série

1 Introduction

Savoir distinguer les besoins de synchronisation

- Exclusion mutuelle :
 - Dès qu'un processus a commencé à exécuter du code qui manipule une ressource partagée, il exclut de fait les autres: ils ne doivent pas pouvoir atteindre la portion de leur code dans laquelle ils utilisent aussi cette même ressource
- Synchronisation *conditionnelle* :
 - Une certaine condition doit être vérifiée pour que l'exécution d'un processus puisse progresser : la condition porte en général sur l'état d'une ressource et on espère qu'une exécution concurrente rende la condition vraie

1 Introduction

Se synchroniser nécessite d'attendre

- Pourquoi attendre ?
 - Exclusion mutuelle : qu'un processus déjà dans une section de code dite « critique » la quitte



- Synchro. Conditionnelle : qu'un processus rende la condition de progression vraie



1 Introduction

Outils de synchronisation

- Doivent permettre de faire attendre
 - L'occurrence d'un événement particulier qui permettrait d'arrêter d'attendre
 - Événement pourrait être notifié par un signal : attente active
 - De préférence de manière passive
 - Sémaphore : disponibilité d'un jeton
 - Variable de condition de Moniteur : réveil car condition change
 - Attente bloquante sur réception de message
- De permettre l'échange d'informations
 - Interaction volontaire = besoin d'échanger une information (en général)
 - Variable partagée, lue ou écrite
 - Fichier dans lequel on lit ou écrit
 - Contenu d'un message émis puis reçu

Attente active

- test « dois-je attendre ? » est exécuté en boucle
 - Ne provoque pas de commutation de contexte volontaire (« je rends la main ») de la part du processus qui attend
 - Sauf que si on veut qu'un processus fasse progresser les choses et donc que l'attente se termine, il faut bien qu'il ait la main, donc, que le SE commute vers lui !
 - Utilise / gaspille du temps CPU (jusqu'à épuisement du quantum de temps dans le cas d'un processus ordonnancé par le SE)
 - A utiliser de préférence pour des attentes qu'on sait être courtes car la condition va changer rapidement ...
 - Ex: ... si la section de code critique est toujours très courte
 - Ex: ... si l'événement qui va rendre la condition vraie arrive très fréquemment
 - et si ce changement peut arriver en parallèle de l'attente
 - Ex: cas d'une machine multi-CPU
 - Ex: événement provoqué par un autre élément que le CPU (ex, fin d'E/S disque)

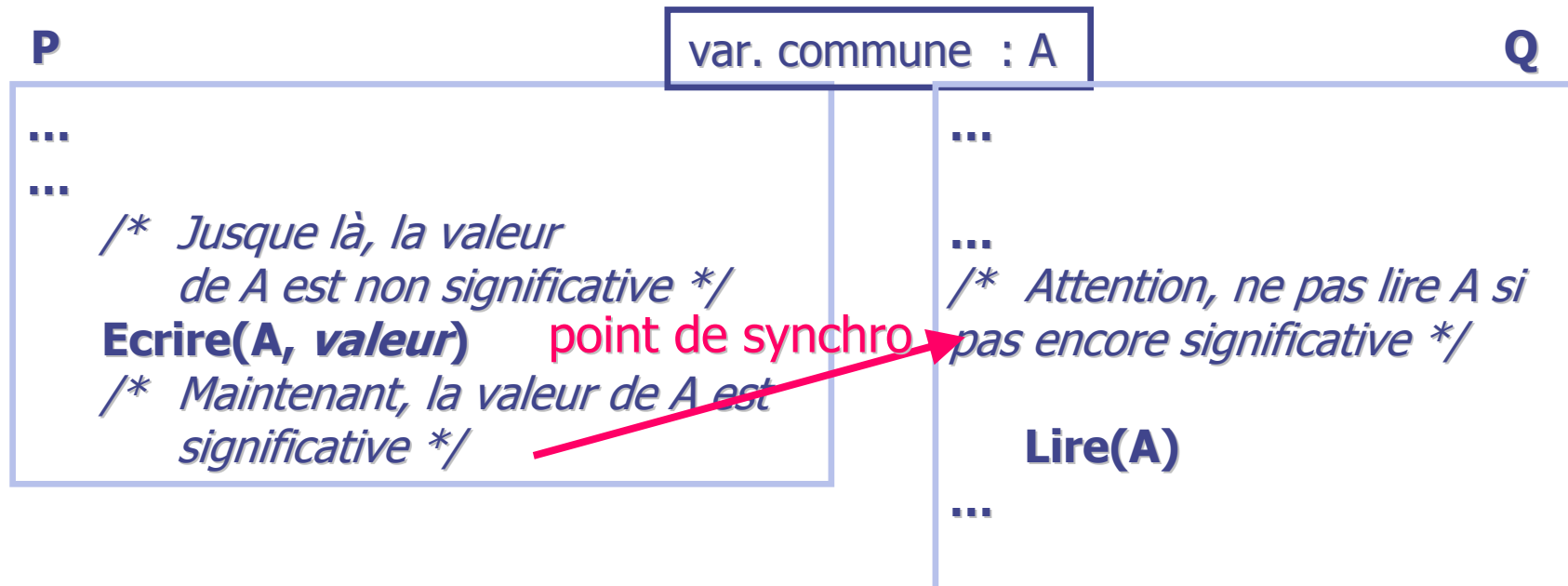
Attente passive

- Le test « dois-je attendre ? » est exécuté et si il rend « Vrai », le processus demande à être bloqué, et fait le pari qu'il sera débloqué de l'extérieur
 1. « rend la main » volontairement
 - immédiatement ... dès qu'un point de commutation est atteint
 - pas de gaspillage du temps CPU
 2. Une autre entité (SE ou un autre processus) prend l'initiative de débloquer (rendre éligible) le processus bloqué car l'attente n'a plus lieu d'être
- Une seule commutation de contexte du processus qui doit attendre
 - C'est « 1 » dans le cas théorique : la condition n'a pas changé, entre la prise de décision de débloquer et le moment où le processus débloqué reprend vraiment son exécution
 - En pratique, il peut donc s'avérer nécessaire de refaire le test
 - Car d'autres processus auront pu passer avant le processus réveillé et avoir rendu la condition d'attente de nouveau vraie

1 Introduction

Illustrations (1/4)

- Besoin de partager de l'information, via une Variable
 - Faire Attendre Q tant que A n'est pas écrite
 - Avant le point de synchro, P et Q pouvaient en être n'importe où de leur exécution
 - Au point de synchro, Q se synchronise avec P



1 Introduction

Illustrations (2/4)

var. commune :

A (fichier, variable, ...qui doit exister et contenir des données, ou avoir une valeur)

A_est_remplie : sémaphore INIT 0

P

Q

...

...

/ Jusque là, peu importe la valeur de A */*

Ecrire(A, valeur)

/ Maintenant, la valeur de A est significative */*

V(A_est_remplie)

...

...

/ Avant d'utiliser A, il faut qu'elle ait une valeur significative */*

P(A_est_remplie)

Lire(A)

...

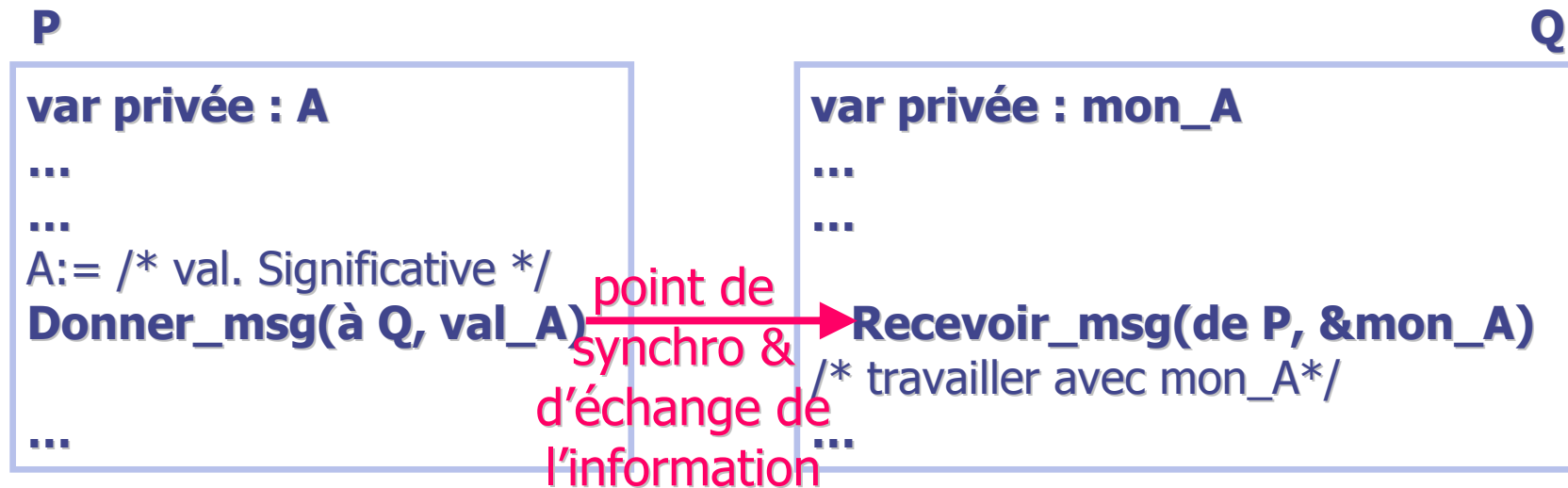
point de synchro



1 Introduction

Illustrations (3/4)

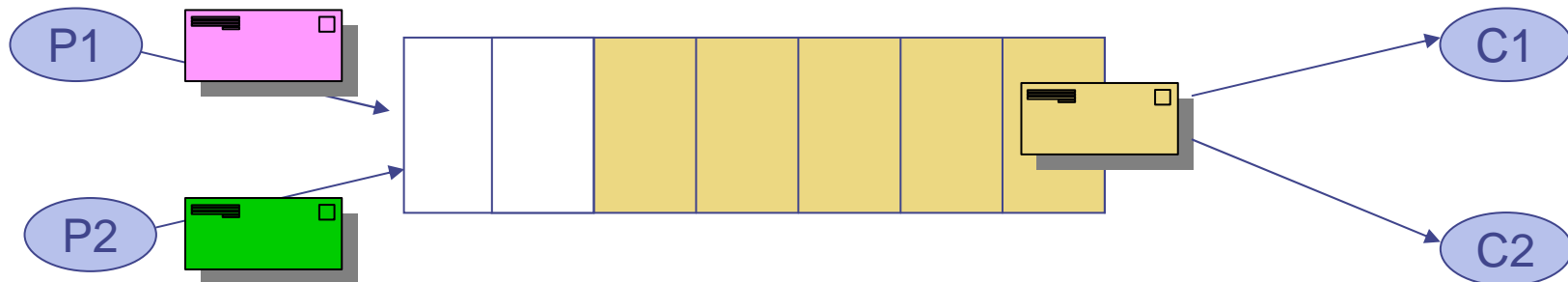
- Par échange de message
 - Réception bloquante tant que le message n'est pas arrivé = synchronisation implicite
 - P doit généralement identifier Q de manière explicite



1 Introduction

Illustrations (4/4)

- Exemple 2 :
 - Une file de messages (une boîte aux lettres), FIFO en général



- Nombreux problèmes de synchro et de communication sous-jacents:
 - Une case libre doit être allouée à un seul processus producteur
 - Que faire si plus de case libre
 - Une case pleine doit être lue par un seul consommateur
 - Que faire si aucune case pleine

1 Introduction

Illustration de synchronisation :

« chez le médecin »

- Le médecin ne peut s'occuper que d'un patient à la fois, si il y en a ! (exception: le cas d'une famille)
- Le cabinet doit être ouvert aux patients
- Pour tout patient, attente dans la salle d'attente (si place) jusqu'à ce que la condition « c'est à mon tour » soit vraie
 - active : je ne dors pas, ex. je lis, sur ma chaise, et je surveille en permanence si ce ne serait pas mon tour; dès que le médecin dit que c'est le tour du patient suivant, je suis en (théorie en) mesure de le voir et de me lever si c'est effectivement mon tour (hypothèse passage en FIFO)
 - passive : je dors sur ma chaise, j'ai confiance, on me réveillera quand ce sera mon tour
 - Variante : est-ce mon tour ou le tour du patient suivant ? Il se peut qu'on me réveille parce que le médecin a dit que c'est le tour du patient suivant; mais, ce n'est peut-être pas mon tour, donc, je vais tester et si besoin, me rendormir

1 Introduction

Solutions au besoin de synchronisation

- Utilisation des outils de synchro et de communication ...
- selon le schéma de synchronisation voulu
 - *Exclusion Mutuelle*
 - Rendez-vous, de 2, ou N entités
 - Lecteur(s) / Rédacteur(s)
 - Producteur(s) / Consommateur(s)
 - variantes plus ou moins complexes (voir TDs)

Coopération entre processus & Synchronisation + Communication

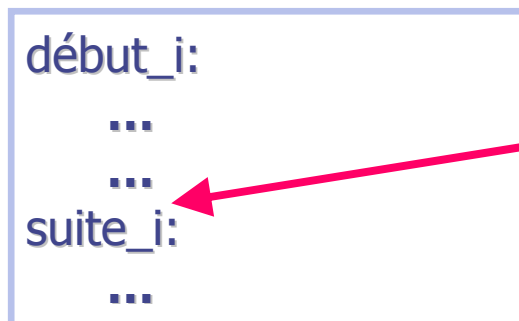
1. Introduction
2. Rendez-vous de N entités
3. Producteur(s) / Consommateur(s)
4. Exemple d'un tube Unix : synchronisation et communication
5. Généralisation Exclusion Mutuelle avec des processus de type Lecteur ou Rédacteur

2 Rendez-vous de N entités

Exposé du problème

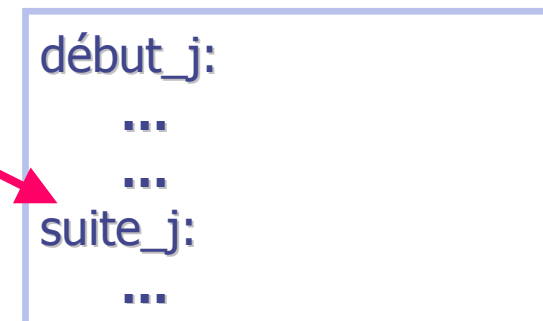
- Etablir un point de rendez-vous dans le déroulement de N processus (N supposé connu)
 - Définition de la contrainte de synchronisation :
Pour tout i, j dans $[1, N]$
 $\text{fin}(\text{début}_i)$ PRECEDE $\text{début}(\text{suite}_j)$

P_i ($1 \leq i \leq N$)



point de synchro :
Le rendez-vous

P_j ($1 \leq j \leq N$)



2 Rendez-vous de N entités

Solution à base de sémaphores

- Programmation d'un réveil en chaîne : version 1

Pi pour tout i

```
début_i:                                TOUS_LA : sémaphore INIT 0
...
nombre_présent_RDV++ /* variable commune */
Si (Nombre_présent_RDV < N) Alors P(TOUS_LA) FinSi
/* Le dernier arrive ici sans bloquer */
nombre_présent_RDV- -
Si (nombre_présent_RDV > 0) Alors V(TOUS_LA) FinSi
suite_i:
...
```

2 Rendez-vous de N entités

Solution à base de sémaphores (2)

- Problème version 1 : Solution incomplète
 - Modification et test de *nombre_présents_RDV* peuvent s'entrelacer
 - Exemple :
 1. P_{n-1} exécute *nombre_présent_RDV++*
 2. P_n exécute *nombre_présent_RDV++*
 3. P_{n-1} teste *nombre_présent_RDV* : il croit être le dernier !
 4. P_n teste *nombre_présent_RDV* : il est aussi le dernier !

=> Exécution d'un V(TOUS_LA) en trop par rapport au nombre de P(TOUS_LA) exécutés

2 Rendez-vous de N entités

Solution à base de sémaphores (3)

- Programmation d'un réveil en chaîne : version 2

Pi pour tout i

```
début_i:                                TOUS_LA : sémaphore INIT 0
...                                       MUTEX : sémaphore INIT 1
P(MUTEX)
nombre_présent_RDV++                    /* variable commune */
Si (Nombre_présent_RDV < N)           Alors
  V(MUTEX)
  P(TOUS_LA)                          /* bloque sans garder MUTEX ! */
P(MUTEX) FinSi
  /* Le dernier arrive ici sans bloquer */
  nombre_présent_RDV- -
  Si (nombre_présent_RDV > 0) Alors
    V(TOUS_LA) FinSi
  V(MUTEX)
suite_i:
...
```

2 Rendez-vous de N entités

Moniteur : description complète

- Moniteur = module
 - Propose des **opérations** qui s'exécutent toujours en exclusion mutuelle
 - Besoin de blocage :
 - Il peut être nécessaire de bloquer une entité « à l'intérieur » du moniteur car une **condition n'est pas satisfaite**
 - Utilisation de **variables de condition**

2 Rendez-vous de N entités

Synchronisation par outil moniteur

- Variable de condition
 - Permet de bloquer une entité « à l'intérieur » du moniteur
 - Ressemble aux sémaphores : manipulée au travers d'une interface précise :
 - Opération WAIT(x:condition) INDIVISIBLE
 - *Ranger contexte entité dans x.File*
 - *mettre entité dans l'état bloqué*
 - Opération SIGNAL(x:condition) ou NOTIFY (e.g. Java), INDIVISIBLE
 - Si (x.File non vide) Alors
 - *retirer de x.File UN processus*
 - *mettre ce processus dans l'état prêt*
 - Rem : Contrairement à V(), SIGNAL() ne laisse pas de trace
 - On peut en faire plus que nécessaire, mais attention aux signaux de réveil « perdus », car pas mémorisés ...

2 Rendez-vous de N entités

Synchronisation par outil moniteur (2)

- Propriétés à respecter dans un moniteur
 - Si une entité se bloque sur une variable de condition à l'intérieur du moniteur, elle doit laisser libre l'accès au moniteur
 - Si une entité à l'intérieur du moniteur en réveille une autre, bloquée sur une variable de condition : cela VIOLE la propriété d'exclusion mutuelle
 - Pour Hoare : L'entité qui exécute SIGNAL() est bloquée jusqu'à ce que l'entité réveillée quitte le moniteur
 - Pour Brinch-Hansen : L'entité ayant exécuté un SIGNAL() doit quitter immédiatement le moniteur

2 Rendez-vous de N entités

Solution à base de Moniteur

- Exemple de Solution : avec réveil en chaîne

```
type RDV : moniteur
  variable nombre_présent_RDV : entier, INIT 0
  variable TOUS_LA : condition

procédure J'arrive_au_RDV      /* En exclusion mutuelle */
début
  nombre_présent_RDV++
  Si (Nombre_présent_RDV < N) Alors
    WAIT(TOUS_LA) FinSi
    /* Le dernier arrive ici sans bloquer */
    SIGNAL(TOUS_LA)
fin

début      /* Initialisation du moniteur */
fin
```

2 Rendez-vous de N entités

Solution à base de Moniteur (2)

- Mise en pratique

var. commune **Mon_RDV** : RDV

Pi pour tout i

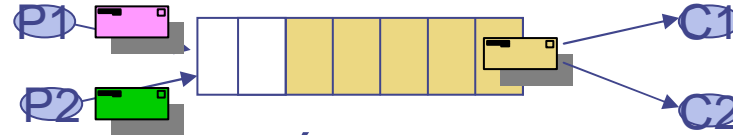
```
début_i:  
  ...  
  Mon_RDV. J'arrive_au_RDV  
  ...  
suite_i:  
  ...
```

Coopération entre processus & Synchronisation + Communication

1. Introduction
2. Rendez-vous de N entités
3. **Producteur(s) / Consommateur(s)**
4. Exemple d'un tube Unix : synchronisation et communication
5. Généralisation Exclusion Mutuelle avec des processus de type Lecteur ou Rédacteur

3 Producteur(s) / Consommateur(s)

Producteurs / Consommateurs avec sémaphore



- Buffer (circulaire) avec nombre borné de places
- 2 Sémaphores pour la synchronisation conditionnelle :
 - Si plus de place disponible, processus type producteur attend
 - Si pas de place remplie, processus type consommateur attend
- Sémaphore pour gérer exclusion mutuelle entre producteurs ou entre consommateurs pour ne pas qu'ils agissent sur la même case
 - Variante: un sémaphore pour le côté des producteurs, et un sémaphore pour le côté des consommateurs
 - On peut démontrer que les producteurs ne peuvent jamais être en train de remplir une case, et en même temps, les consommateurs en train de vider cette même case

3 Producteur(s) / Consommateur(s)

Synchronisation Prod/Cons avec sémaphore, échange info. via buffer

```
int bufferSz;  
Msg[] buffer = new Msg[bufferSz];  
Semaphore mutexIn = new Semaphore(1);  
Semaphore mutexOut = new Semaphore(1);  
  
Semaphore notFull = new Semaphore(bufferSz);  
Semaphore notEmpty = new Semaphore(0);
```

```
Produce(Msg msg) {  
// if buffer is full, wait for one empty entry  
notFull.P();  
mutexIn.P();  
buffer[in] = msg;  
in = in + 1 % bufferSize;  
mutexIn.V();  
// wakeup some waiting process  
notEmpty.V();  
}
```

```
Msg Consume() {  
// if buffer is empty, wait for one item  
notEmpty.P();  
mutexOut.P();  
Msg msg = buffer[out];  
out = out + 1 % bufferSz;  
mutexOut.V();  
notFull.V();  
return msg;  
}
```

Coopération entre processus & Synchronisation + Communication

1. Introduction
2. Rendez-vous de N entités
3. Producteur(s) / Consommateur(s)
4. Exemple d'un tube Unix : synchronisation et communication
5. Généralisation Exclusion Mutuelle avec des processus de type Lecteur ou Rédacteur

5 Lecteurs / Rédacteurs

Exposé du problème

- Généralisation de l'usage exclusif d'une ressource lorsque l'on veut distinguer :
 - Usage en **modification (en écriture/rédaction)**
 - **exclusivité stricte** : 1 seul à la fois
 - Usage en **consultation (en lecture)**
 - **exclusivité relâchée** : éventuellement plusieurs ...
- Distinction **exclusivité stricte / relâchée** :
 - Exclusion mutuelle entre rédacteurs : **OUI**
 - Exclusion mutuelle entre lecteurs et rédacteurs : **OUI**
 - Exclusion mutuelle entre lecteurs : **NON**
- Différentes politiques possibles afin de gérer les priorités pour les différents types de processus

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre

- First-Come-First-Served (FIFO)
 - Exemple de ressource : un fichier (plutôt qu'une variable, une ressource physique, ou autre)

var. commune Mutex_fichier : sémaphore INIT 1

Lecteur ou Rédacteur

```
...  
P(Mutex_fichier)  
Lire ou Ecrire ...  
V(Mutex_Fichier)  
...
```

- Aucun accès simultané des lecteurs (exclusion mutuelle classique !)
- Par contre, c'est l'ordre de passage souhaitable car le plus équitable

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (2)

- Politique favorisant les lecteurs (avec sémaphore)

```
var. commune  Mutex_fichier : sémaphore INIT 1
              Mutex_nbl   : sémaphore INIT 1
              Nb_lect     : entier INIT 0
```

Rédacteur

```
...
P(Mutex_fichier)
Ecrire ...
V(Mutex_Fichier)
...
```

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (3)

- Politique favorisant les lecteurs (sémaphore : suite...)

Lecteur

```
P(Mutex_nbl)
Nb_lect++
Si (Nb_lect == 1 ) Alors           /* premier lecteur : obtenir fichier */
    P(Mutex_fichier) FinSi
V(Mutex_nbl)
Lecture ...
P(Mutex_nbl)
Nb_lect - -
Si (Nb_lect == 0 ) Alors         /* dernier lecteur : relâcher fichier */
    V(Mutex_fichier) FinSi
V(Mutex_nbl)
...
```

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (4)

- Politique favorisant les lecteurs (avec moniteur)

```
var. commune lire_ecrire : Synchro_Lire_Ecrire
```

Rédacteur

```
...  
lire_ecrire.début_écriture  
Ecrire ...  
lire_ecrire.fin_écriture  
...
```

Lecteur

```
...  
lire_ecrire.début_lecture  
Lire ...  
lire_ecrire.fin_lecture  
...
```

```
type Synchro_Lire_Ecrire : moniteur  
variable nb_lect : entier, INIT 0  
variable écriture_en_cours : booléen, INIT FAUX  
variable accord_lecture, accord_écriture : condition
```

...

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (5)

- Politique favorisant les lecteurs (avec moniteur): suite

```
type Synchro_Lire_Ecrire : moniteur (suite)
```

```
  procédure début_écriture
```

```
    Si (nb_lect >0 OU écriture_en_cours)
```

```
      Alors Wait(accord_écriture)
```

```
      FinSi
```

```
      écriture_en_cours := VRAI
```

```
  procédure fin_écriture
```

```
    écriture_en_cours := FAUX
```

```
    Si (accord_lecture NON VIDE)
```

```
      Alors Signal(accord_lecture) /* une lecture en attente n'attend que la fin  
de l'écriture en cours */
```

```
      Sinon Signal(accord_écriture)
```

```
      FinSi
```

...

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (6)

- Politique favorisant les lecteurs (avec moniteur): suite

```
type Synchro_Lire_Ecrire : moniteur (suite)

  procédure début_lecture
    Si (écriture_en_cours OU accord_écriture NON VIDE)
      Alors Wait(accord_lecture) FinSi
      nb_lect ++
      Signal(accord_lecture)      /* Réveil en chaîne ... */

  procédure fin_lecture
    nb_lect - -
    Si (nb_lect == 0)
      Alors Signal(accord_écriture)
    FinSi
```

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (7)

- Politique équitable (avec sémaphores)
 - Aussi équitable que FIFO,
 - Mais, en autorisant passage parallèle et non sérialisé des lecteurs

var. commune Mutex_fichier : sémaphore INIT 1
 Mutex_nbl : sémaphore INIT 1
 Ordre_arrivée : sémaphore INIT 1
 Nb_lect : entier INIT 0

Rédacteur

```
...  
P(Ordre_arrivée)  
P(Mutex_fichier)  
Ecrire ...  
V(Mutex_Fichier)  
V(Ordre_arrivée)  
...
```

5 Lecteurs / Rédacteurs

Politiques de mise en œuvre (8)

- Politique équitable (avec sémaphores : suite)

Lecteur

```
P(Ordre_arrivée)
P(Mutex_nbl)
Nb_lect++
Si (Nb_lect == 1 ) Alors           /* premier lecteur : obtenir fichier */
    P(Mutex_fichier) FinSi
V(Mutex_nbl)
V(Ordre_arrivée)           /* Pour faire venir avec moi tous les lecteurs */
Lecture ...
P(Mutex_nbl)
Nb_lect - -
Si (Nb_lect == 0 ) Alors         /* dernier lecteur : relâcher fichier */
    V(Mutex_fichier) FinSi
V(Mutex_nbl)
```