Attente passive

- verrou
- sémaphore
- moniteur

Verrou (1)

- élément de synchronisation à base d'attente passive
- attributs
  - état : ouvert/fermé
  - enAttente : liste d'activités en attente devant le verrou fermé
- interface d'utilisation – opérations système atomiques
  - *verrouiller* – acquérir le verrou si disponible, sinon se bloquer
  - *déverrouiller* – libérer le verrou et réveiller d'autres activités en attente sur le verrou

Verrou (2)

```
verrouiller (verrou v)
    si v.état = ouvert alors v.etat = ferme
    sinon
        bloquer l'activité (la rajouter à v.enAttente)
    fsi

déverrouiller (verrou v)
    si v.enAttente non vide alors
        débloquer une activité en attente
    sinon
        v.état = ouvert
    fsi
```

initialisation du verrou
état (ouvert/fermé)
file d'attente vide

Verrou (3)

- outil intuitif, simple
- implémentation de la synchronisation compétitive facile
  - verrou *lock*, initialement ouvert

```
verrouiller(lock)
      SC
déverrouiller(lock)
```

Sémaphore

- accès aux ressources en nombre fini (généralisation du mécanisme de verrou) – opérations atomiques
- Edsger Dijkstra (1965) – objet constitué d'un entier positif et d'une file d'attente des processus bloqués
  - `Init(S, val)` = initialise la valeur de l'entier
  - `P(S)` (Proberen) = puis-je
  - contrôle d'autorisation + blocage éventuel du demandeur
  - `V(S)` (Verhogen) = vas-y
  - ajout d'une autorisation + déblocage éventuel d'un demandeur
- initialisé au nombre de ressources disponibles

Implémentation des sémaphores

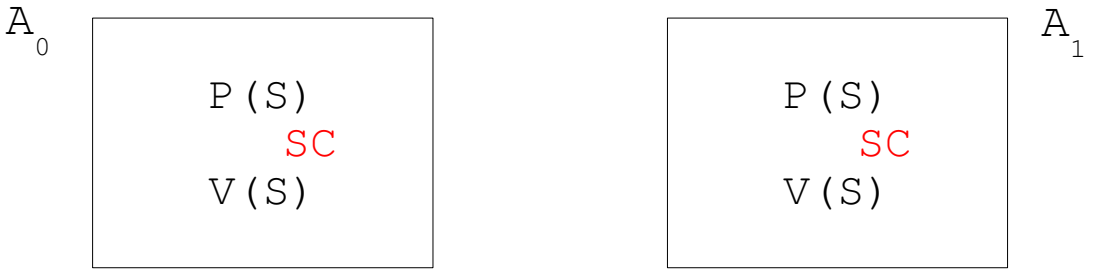
```
P(S)
    S.value--
    si S.value < 0 alors
        bloquer l'activité
        ajouter cette activité à la file d'attente
    fsi

V(S)
    S.value++
    si S.value ≤ 0 alors
        retirer l'activité A de la file d'attente
        réveiller l'activité A
    fsi
```

```
Init(S,val)
    S.value ← val
    file d'attente ← vide
```

Exclusion mutuelle avec les sémaphores

- sémaphore S binaire (initialisé à 1) = *mutex*



Sémaphore – exemple

- gestion des places d'un parking
- `Init` : sémaphore (nombre de places) `nbPl`
- entrée parking : `P(nbPl)`
  - décrémente la valeur (nombre de places libres)
  - mise en attente si valeur < 0
- sortie parking : `V(nbPl)`
  - incrémente la valeur
  - ne dépasse jamais la valeur d'initialisation

Sémaphores – problèmes

- utilisation incorrecte des opérations
  - V(S) … V(S)
  - P(S) … P(S)

  } sémaphores binaires

  - P(S1) P(S2) / P(S2) P(S1)
- oubli des opérations P/V

Moniteur

- outil de haut niveau implémentant un mécanisme efficace pour la synchronisation (Hoare 1974)
- module comprenant
  - des données (partagées en accès concurrent)
  - des procédures (des sections de code critique) – exécutées en exclusion mutuelle
  - des variables de type condition (verrou conditionnel)
- au maximum une seule activité peut détenir le moniteur → synchronisation à la charge du compilateur

Moniteur - schéma

```
monitor <monitor-name>
    var < shared variables +
       conditions declarations>
    procedure P1(…) finproc
    procedure P2(…) finproc
    …
    procedure Pn(…) finproc
    { initialization code }
fin monitor
```

variables partagées

variable condition = *verrou conditionnel*

- détient un identificateur, mais pas de valeur
- ne doit pas être initialisée
- ne peut être manipulée que par les primitives spécifiques
- représentée par une file d'attente des activités bloquées sur la même condition

Verrou conditionnel

- verrou conditionnel : c
- à chaque verrou conditionnel est associée une file des activités bloquées (FIFO)
- *wait* (*c*)
  - bloque l'activité *A* qui l'utilise et la place en attente
- *signal* (*c*)
  - réveille une activité en attente dans la file des activités bloquées associée au verrou *c*
  - le signal n'est pas mémorisé

$\Rightarrow$ implicitement : exclusion mutuelle

Moniteur – exemple

- exclusion mutuelle

```
monitor moniteurSC
        var pris : booléen
                x : condition

        procedure lock()
                si pris alors
                        wait(x)
                fsi
                pris = vrai
        finproc

        procedure unlock()
                pris = faux
                signal(x)
        finproc

        { initialisation }
        pris = faux
fin monitor
```

A_i

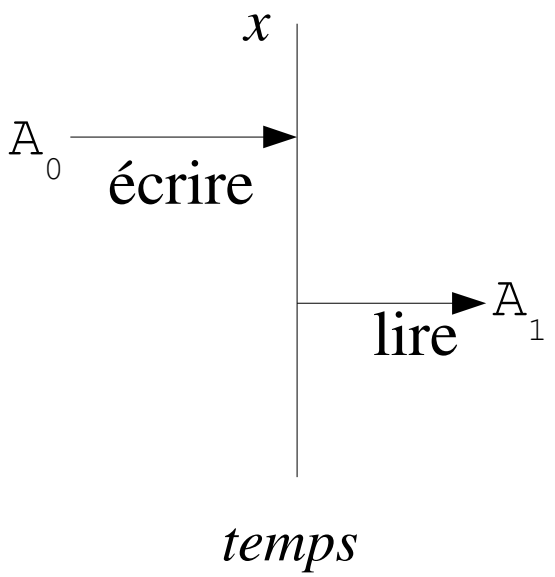
```
moniteurSC.lock()
   SC
moniteurSC.unlock()
```

Problématique : synchronisation coopérative

- coordination de l'exécution des activités (en dehors de leur ordonnancement)
- deux ou plusieurs activités asynchrones s'attendent réciproquement pour qu'un certain événement se produise
- exemple : barrière de synchronisation (gestion de rendez-vous)

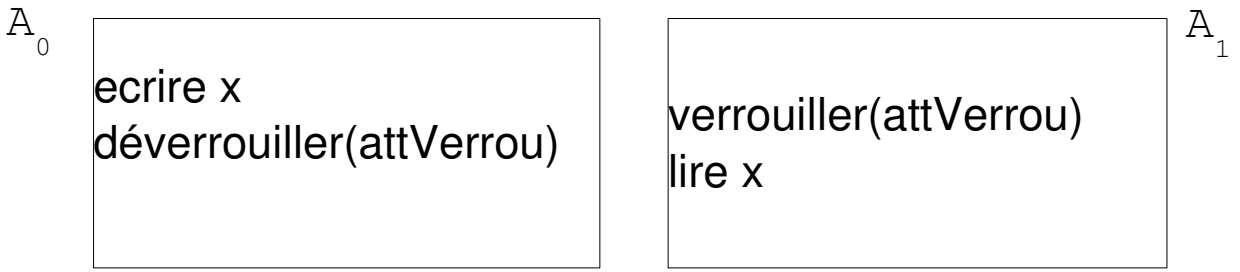
Problème de coordination

- exemple particulier : une activité A_1 peut lire une donnée x seulement après avoir été modifiée par une activité A_0



Problème de coordination – verrou

verrou *attVerrou*
initialisé à fermé



Problème de coordination – sémaphore

sémaphore *attSem*
initialisé à 0

Problème de coordination – moniteur

```
monitor sync
        var fait : booléen
            fini : condition

        procedure fin-ecrire()
                fait = vrai
                signal(fini)
        finproc

        procedure debut-lire()
                si non fait alors
                        wait(fini)
                fsi
                fait = faux
        finproc

        { initialisation }
        fait = faux
fin monitor
```

A_0

```
ecrire x
sync.fin-ecrire()
```

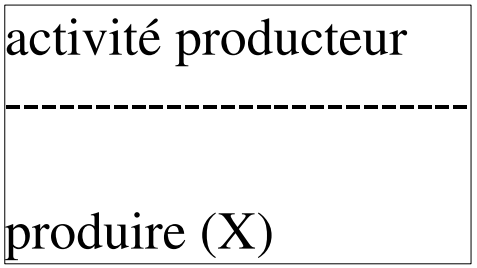
```
sync.debut-lire()
lire x
```

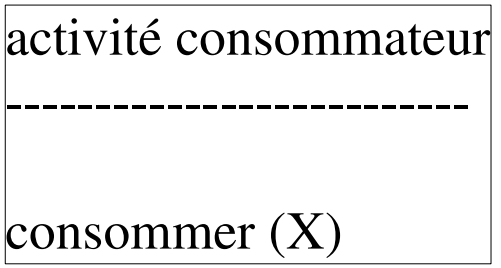
A_1

Problèmes classiques de synchronisation

producteur/consommateur
lecteurs/rédacteurs
salon de coiffure
dîner des philosophes

Producteur/consommateur





- contraintes
  - produire avant consommer
- solutions
  - drapeau booléen estProduit (changement d'état du buffer)
  - verrou

Producteur/consommateur - drapeau

- drapeau booléen estProduit
  - initialisé à faux
- attente active

```
activité producteur
-------------------------

produire (X)
estProduit = vrai
```

```
activité consommateur
-------------------------

tant que non estProduit faire
   ??
fait

consommer (X)
```

Producteur/consommateur - verrou

- verrou commun - *vProd*
- attention aux déverrouillages sans verrouillages : donnée produite

activité producteur
produire (X)
déverrouiller(vProd)

activité consommateur
verrouiller(vProd)
consommer (X)

Producteur/consommateur – données multiples (1)

```
activité producteur
-----------------------
faire
        produire (X)
tant que ...
```

```
activité consommateur
-----------------------
faire
        consommer (X)
tant que ...
```

- gestion des données : buffer
- contrainte en cas de buffer de taille suffisamment grande (buffer infini)
  - la consommation nécessite un buffer non vide
- 2 contraintes en cas de buffer fini
  - la consommation nécessite un buffer non vide
  - la production nécessite un buffer non plein

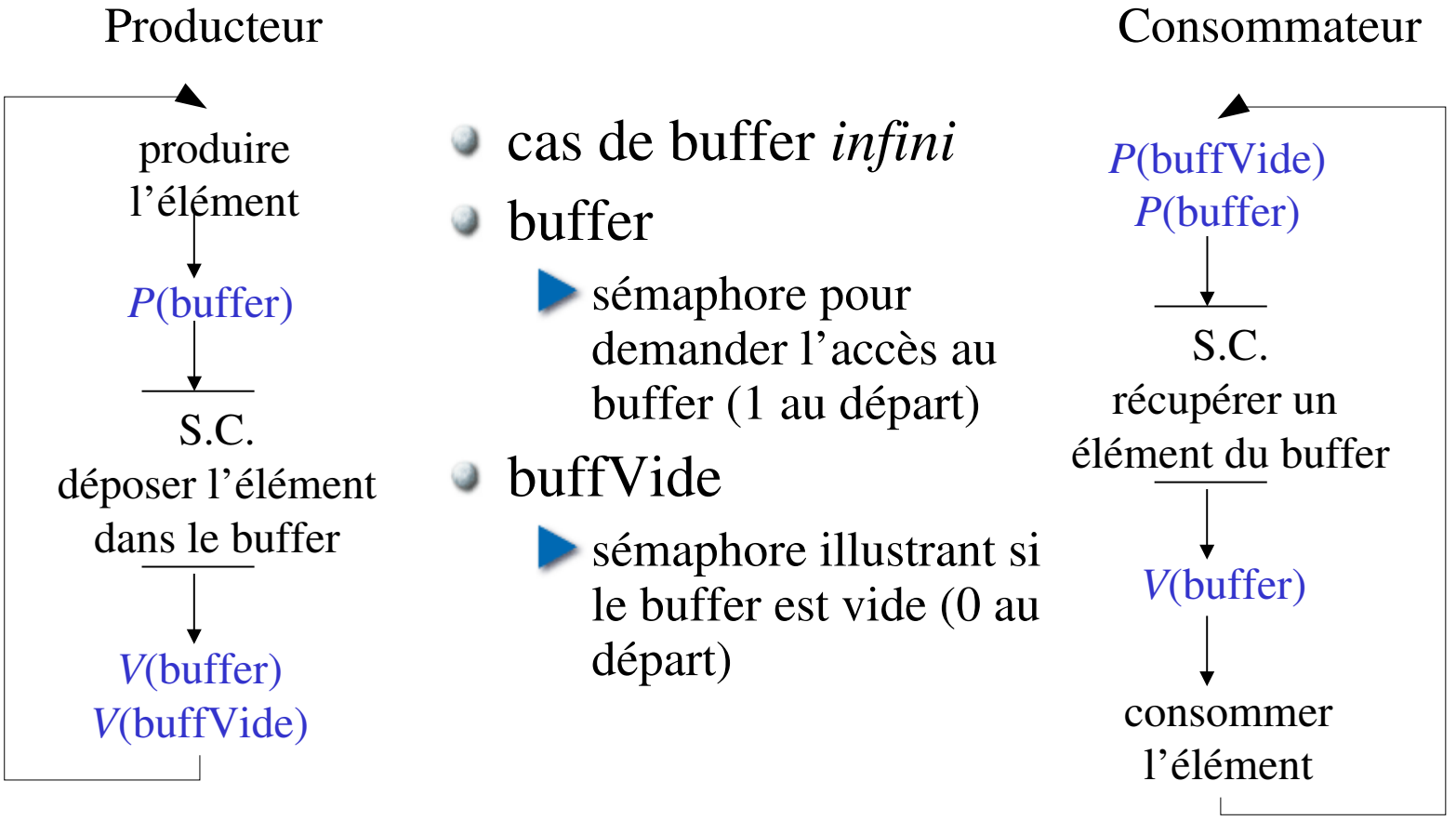
Producteur/consommateur – données multiples (2)

```
activité producteur
-------------------------
faire
        produire (X)
        buffer[ecri]=X
        ecri++
        count++
        // avertir – un élément de plus
tant que ...
```

```
activité consommateur
-------------------------
faire
        // attente si pas d'élément
        X=buffer[lect]
        consommer (X)
        lect++
        count--
tant que ...
```

- buffer : avancement lecture : *lect* / avancement écriture : *ecri*
- attentes : buffer vide (+ buffer plein), accès modification de buffer

Producteur/consommateur – données multiples (3)



- cas de buffer *infini*
- buffer
  - sémaphore pour demander l'accès au buffer (1 au départ)
- buffVide
  - sémaphore illustrant si le buffer est vide (0 au départ)

Salon de coiffure

- type d'accès indifférencié
- sémaphore
  - pris à l'entrée
  - relâché à la sortie

activité client

P(salonCoiff)
entrée salon
coiffure
sortie salon
V(salonCoiff)

Lecteurs/rédacteurs

- types d'accès différents
  - lecture : accès simultanés autorisés
  - écriture : accès unique autorisé
- gestion avec verrou ok mais pas optimale
  - plusieurs lecteurs, priorités
- primitives spécifiques
  - RWLock : read / write

Lecteurs / rédacteurs

Données
---
sémaphore lecture : semL (mutex)
sémaphore écriture : semE (mutex)
entier nombre lecteurs : nbLect = 0

Lecture

P(semL)
nbLect++
si nbLect = 1 alors
    P(semE) fsi
V(semL)
lecture
P(semL)
nbLect--
si nbLect = 0 alors
    V(semE) fsi
V(semL)

Ecriture

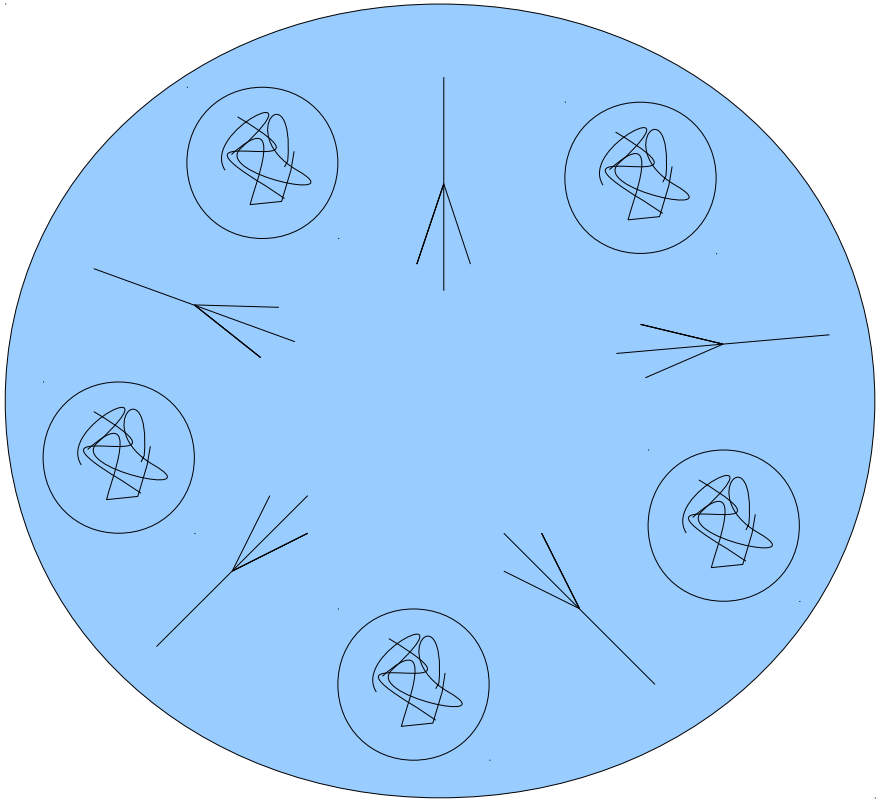
P(semE)
écriture
V(semE)

- garantir l'atomicité = verrou (nbLect, semE, etc.)
- propriété famine : dépend des priorités accordées aux lecteurs/rédacteurs

Dîner des philosophes

1 table

5 fourchettes



5 assiettes de pâtes

- 5 philosophes
  - boucle
    - penser
    - manger
  - besoin de 2 fourchettes pour manger (de gauche et de droite)

Recommandations

- éviter le partage de données
  - pas de synchro
- privilégier la simplicité: accès ressources 1/1
- utiliser les outils à bon escient
- revenir à un schéma classique
- multiplier les circonstances de test
  - exemple de l'anneau
- faire usage des outils

Cas d'étude : Java