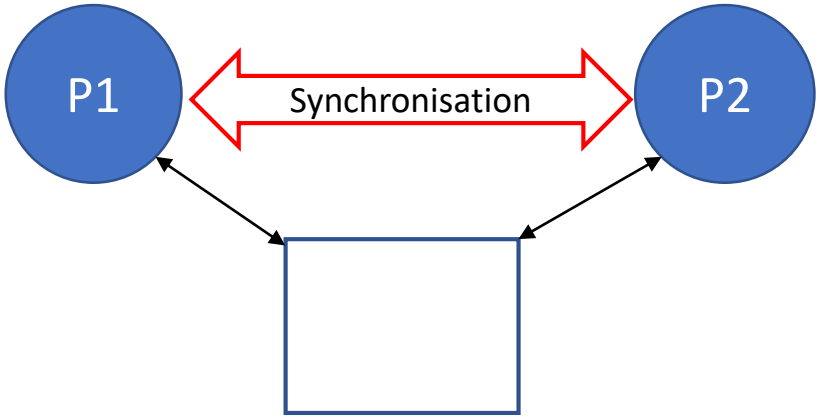
Cours 5 :
Synchronisation des Processus

Synchronisation des processus

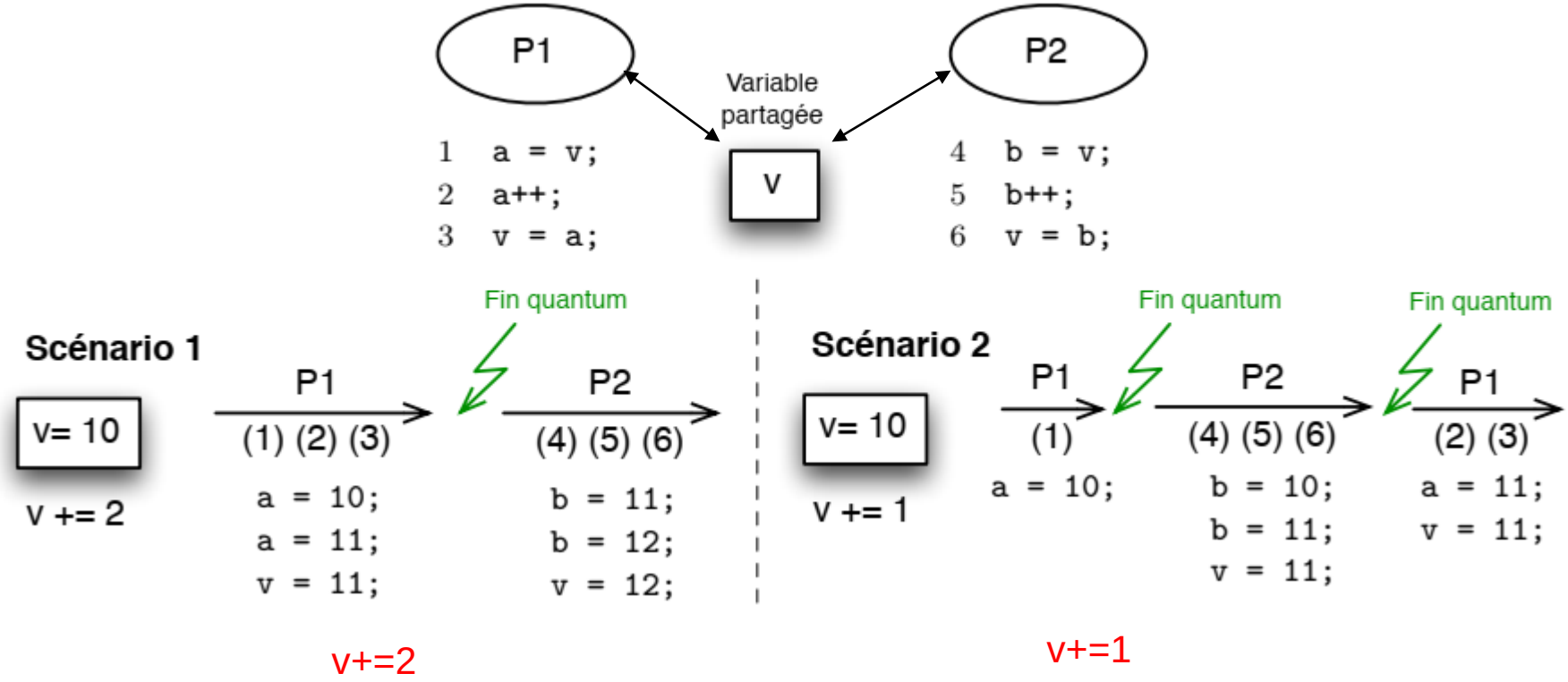
- Processus concurrents partagent souvent des ressources
  - Exemple: variables, fichiers, etc.

- L'accès concurrent à des données partagées peut conduire à des **incohérences** des données.

- Les mécanismes de synchronisation assurent une exécution ordonnée des processus assurant la cohérence



Contexte - Exemple



Définitions

- Ressource critique

  Ressource partagée entre plusieurs processus

  ex :variable *v*

- Section critique (SC)

  Portion de code manipulant une/des ressources critiques

  SC doit être exécutée de manière *indivisible*
  ex: Lignes {(1) (2) (3)} et {(4) (5) (6)}

- Indivisibilité

  Deux SCs sur une même ressource critique ne sont jamais exécutées en parallèle

Définitions

- Synchronisation
  - Opérations qui influent sur l'avancement d'un ensemble de processus

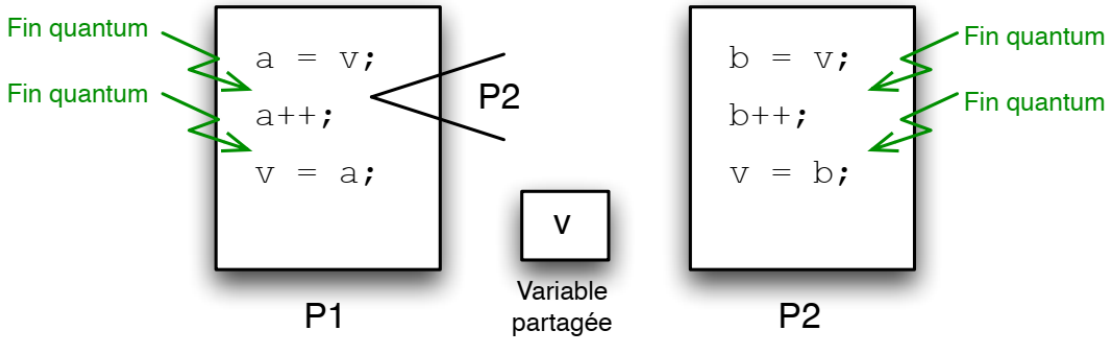
- Exclusion mutuelle
  - Mécanisme de contrôle d'accès à une SC
  - Cas particulier de synchronisation
  - Assure exclusivité d'accès à ***un seul*** processus
    - à tout moment, au plus un processus en SC

Définitions

- Attente active
  - Exécution répétitive d'une primitive de synchronisation
  - Jusqu'à réalisation d'une condition de synchronisation
    - Mobilisation du processeur !
- Interblocage (Deadlock)
  - Attente mutuelle entre deux processus concurrents
    - Blocage définitif !
- Famine
    - Un processus attend indéfiniment pour entrer en section critique.

Exclusion Mutuelle

```
EntrerSC( );
/* Instructions de la SC */
SortirSC( );
```

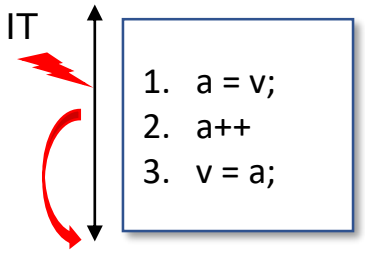


Propriétés

- Sûreté - Au plus un processus en SC
- Vivacité - Toutes les demandes d'accés à la SC sont satisfaites

Solution 1 : Masquage IT Horloge

- Désactivation du temps partagé pendant la SC
  - `EntrerSC:` masque l'IT Horloge
  - `SortirSC:` démasque l'IT Horloge



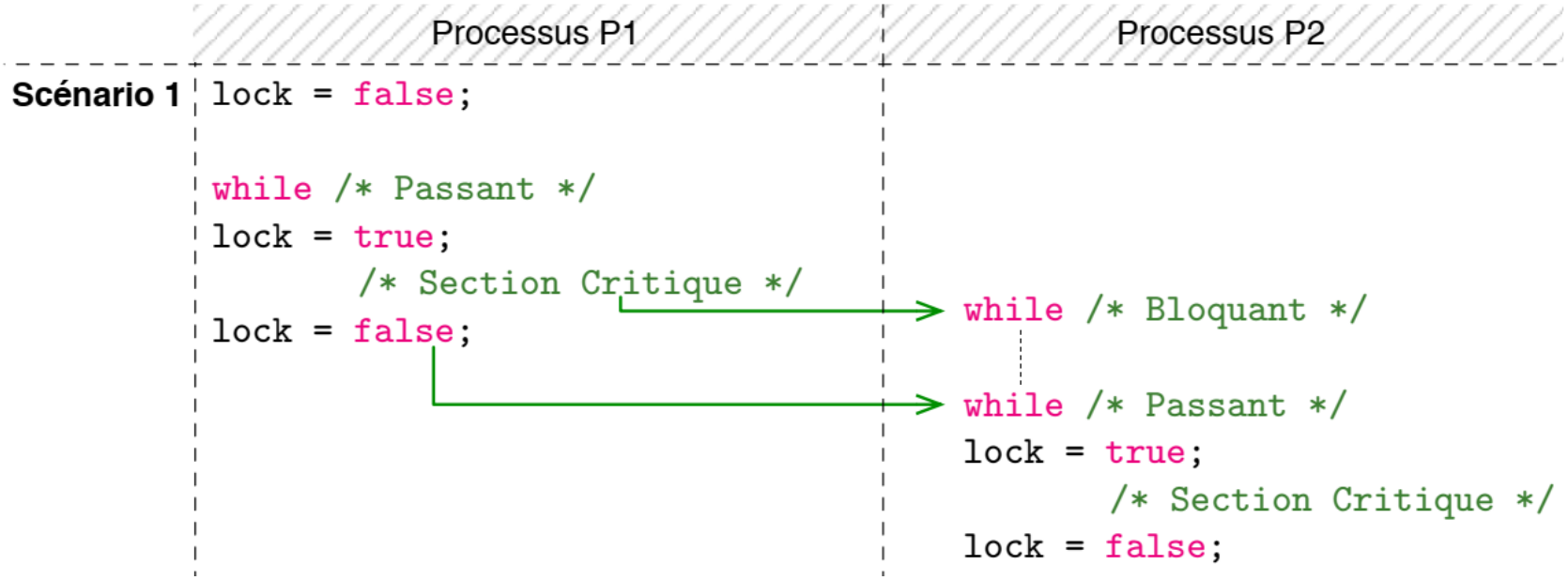
- Mauvaise solution
  - Risque de monopoliser le processeur
  - Exclut tous les processus

Variables de synchronisation

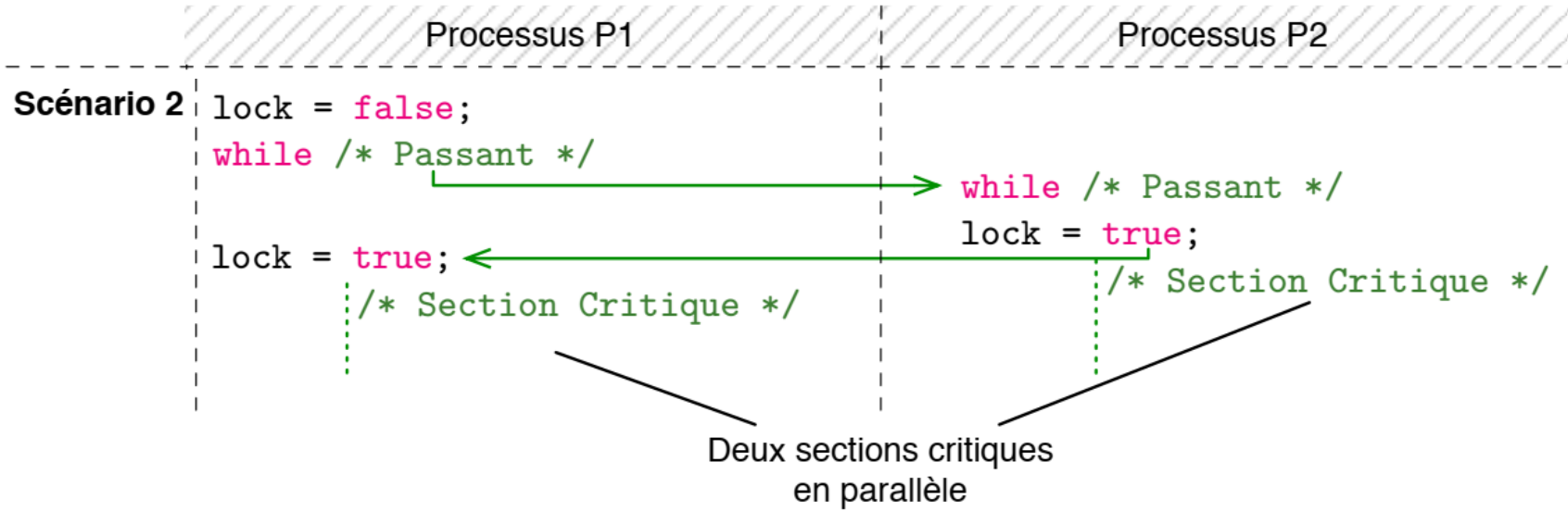
- Utilisation de **variables de synchronisations** et d'**attente active**
- Variable de synchronisation : variable partagée utilisée pour savoir si une ressource critique est disponible
- **Algorithme 1 :**
  - 1 variable booléenne `lock`
    - `true` si la ressource est verrouillée
    - `false` si la ressource est disponible

```
                bool lock = false;

EnterSC()      while(lock == true);
Attente active lock = true;

                /* Section Critique */

SortirSC()     lock = false;
```

Algorithme 1 : exécution 1



~~Algorithme~~ 1 : exécution 2



Ne marche pas !

Algorithme 2 : Algorithme de Peterson

- 2 Processus P_i et P_j , i = 0, j = 1 (généralisable à N processus)
- Variables partagées

bool **flag**[2] = {false,false}; /* flag[i] = true => P_i demande */

int **tour**; /* fixe arbitrairement le processus si les 2 sont demandeurs */

- Principe
  - Pour entrer en SC, P_i positionner flag[i] à true
    puis affirmer en suite que c'est au tour de P_j d'entrer dans sa SC.
  - Si P_i et P_j tentent d'entrer en même temps en SC, la valeur dans tour indiquera le processus autorisé.

Algorithme de Peterson

```
bool flag[2] = {false, false};
int tour;

EnterSC()
Attente active
flag[i] = true;
tour = j;
while(flag[j] == true && tour == j);

/* Section Critique */

SortirSC()
flag[i] = false;
```

Algorithme de Peterson : exécution 1

- P_i demande seul
- flag[i] = true et flag[j] = false

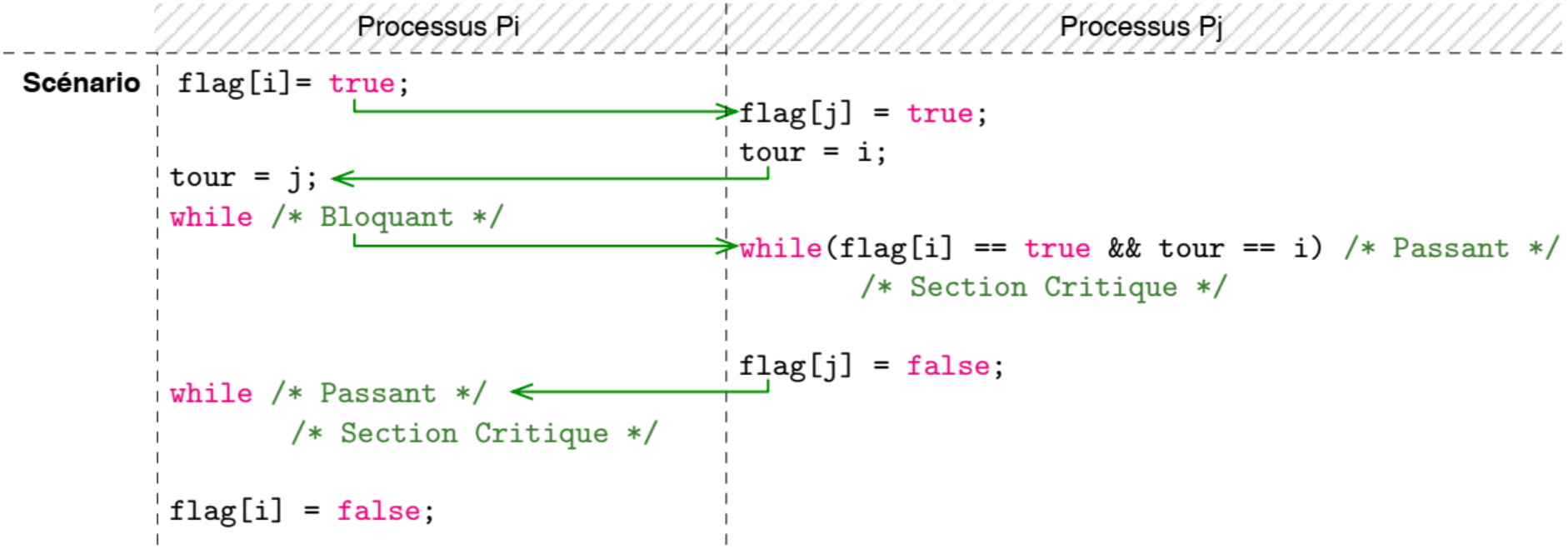
⇒condition du "while" flag[j] == true && tour == j *faux*

⇒ while "passant"

⇒ P_i entre en SC

Algorithme de Peterson : exécution 2

- P_i et P_j demandent => tour fixe l'ordre



Algorithme 3 : test-and-set

- 1 instruction **test-and-set** indivisible : positionne une variable à 1 (true) et retourne son ancienne valeur

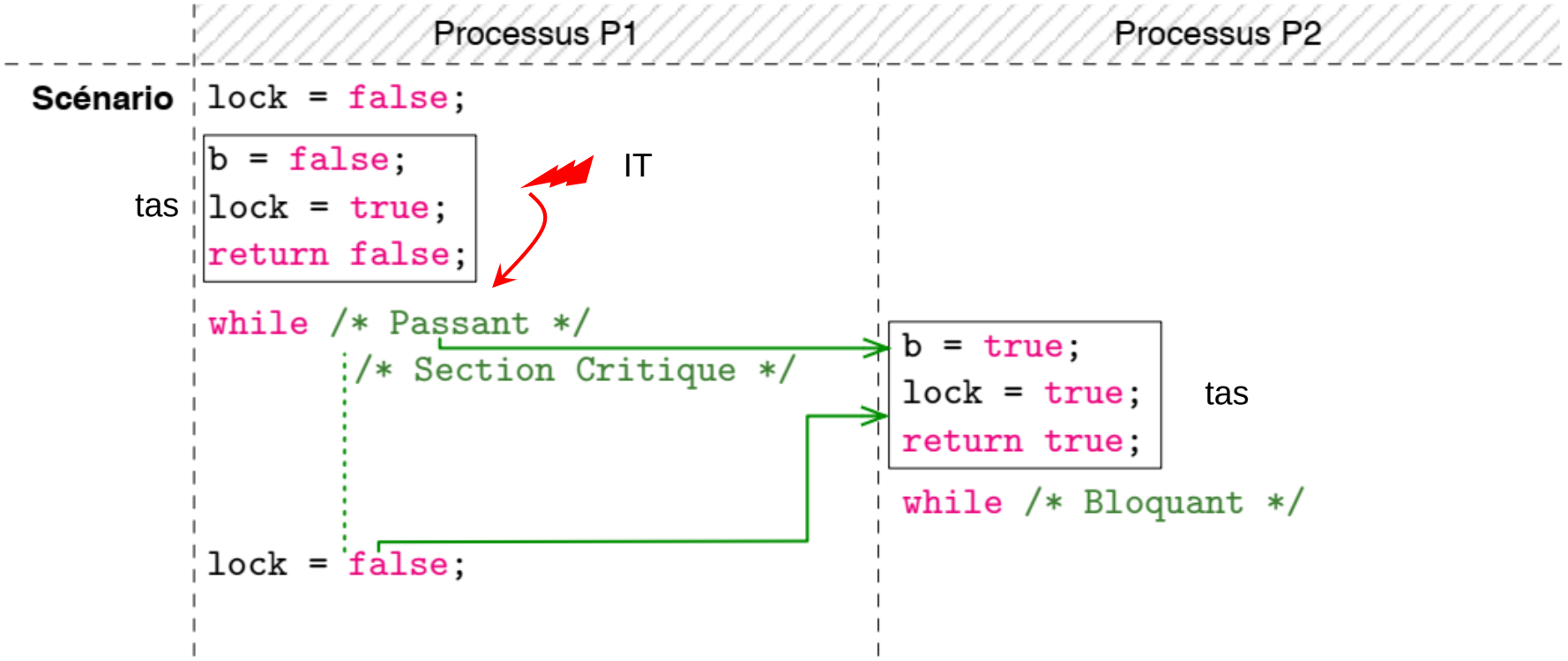
Bloc indivisible
(ie. Pas d'interruption
pendant son exécution)

```
bool tas(bool v) {
        bool b = v;
        v = true;
        return b;
}
```

Pseudo-code test-and-set

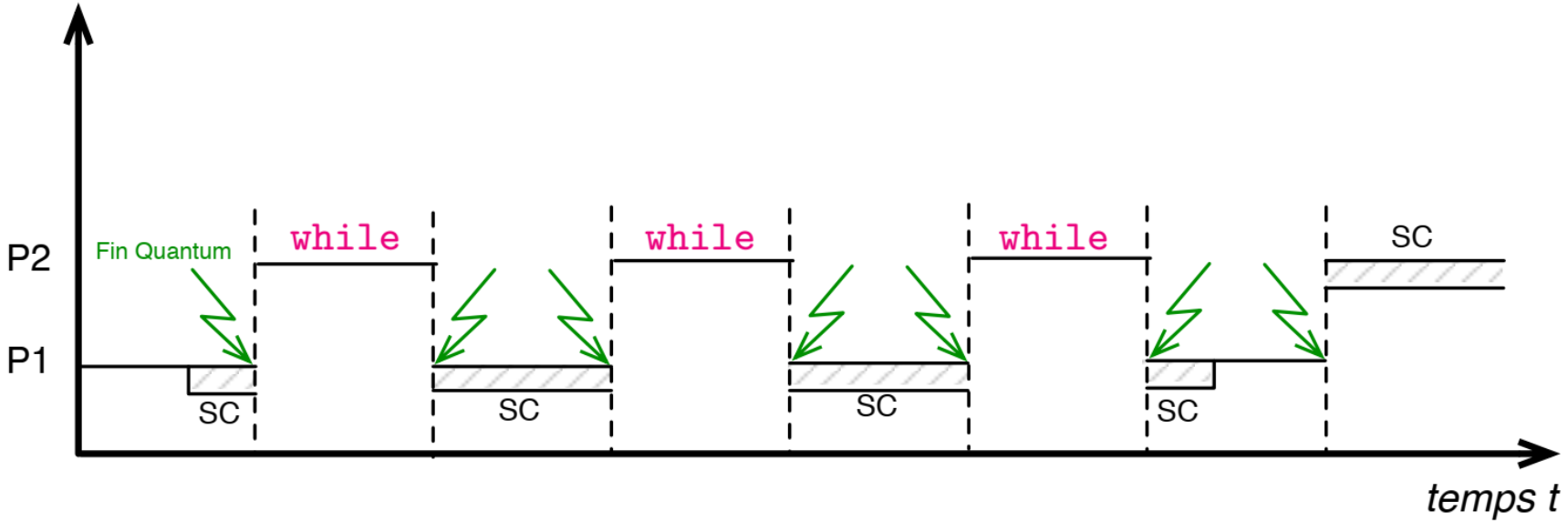
```
        bool lock = false;

EnterSC()
Attente active  while(tas(lock) == true);

                /* Section Critique */

SortirSC()  lock = false;
```

test-and-set : exécution



Limite des approches à base d'attente active

- Algorithme Peterson et test-and-set assure l'exclusion mutuelle
- Problème de performance lié à l'attente active (boucle de test)



Sémaphores

- Sémaphores – Outils de synchronisation
- Introduit par Dijkstra en 1974
- Pas d'attente active
- Sémaphore ***S*** est une variable possédant
  - un ***compteur*** : nombre de ressources disponibles
  - une ***file*** de processus en attente sur S

wikipedia

- L'accès à ***S*** se fait en utilisant les opérations *P* et *V*
  - P : obtenir une ressource : **P**uis-je ? (Proberen – Tester)
  - V: libérer la ressource : **V**as-y ! (Verhogen – Incrémenter)

Sémaphores - Opérations

3 opérations *indivisibles*

- **Init**(sem, val) : Création d'un sémaphore *sem* dont le compteur est initialisé à val.

- **P**(sem) Demande d'acquisition d'une ressource. Si aucune ressource n'est disponible, le processus est bloqué. Décrémente le compteur du sémaphore *sem*

- **V**(sem) Libération d'une ressource. Si la file d'attente n'est pas vide, un processus est débloqué. Incrémente le compteur du sémaphore *sem*.

Sémaphore – Pseudo-code

```
typedef struct {
      int cpt;
      LIST file;
} SEM;


Init(SEM s, int val) {
      s.cpt = val;
      s.file = VIDE;
}
```

```
void P(SEM s) {
      s.cpt --;
      if (s.cpt < 0) {
            inserer(processus_courant, s.file)
            sleep(); /* bloquer */
      }
}

void V(SEM s) {
      s.cpt ++;
      if (s.cpt <= 0) {
            p =retirer(s.file)
            wakeup(p); /* reveiller p*/
      }
}
```

cpt ≥ 0 : cpt correspond au nombre de processus autorisés à accéder à la section critique
cpt < 0 : |cpt| est le nombre de processus bloqués dans la file

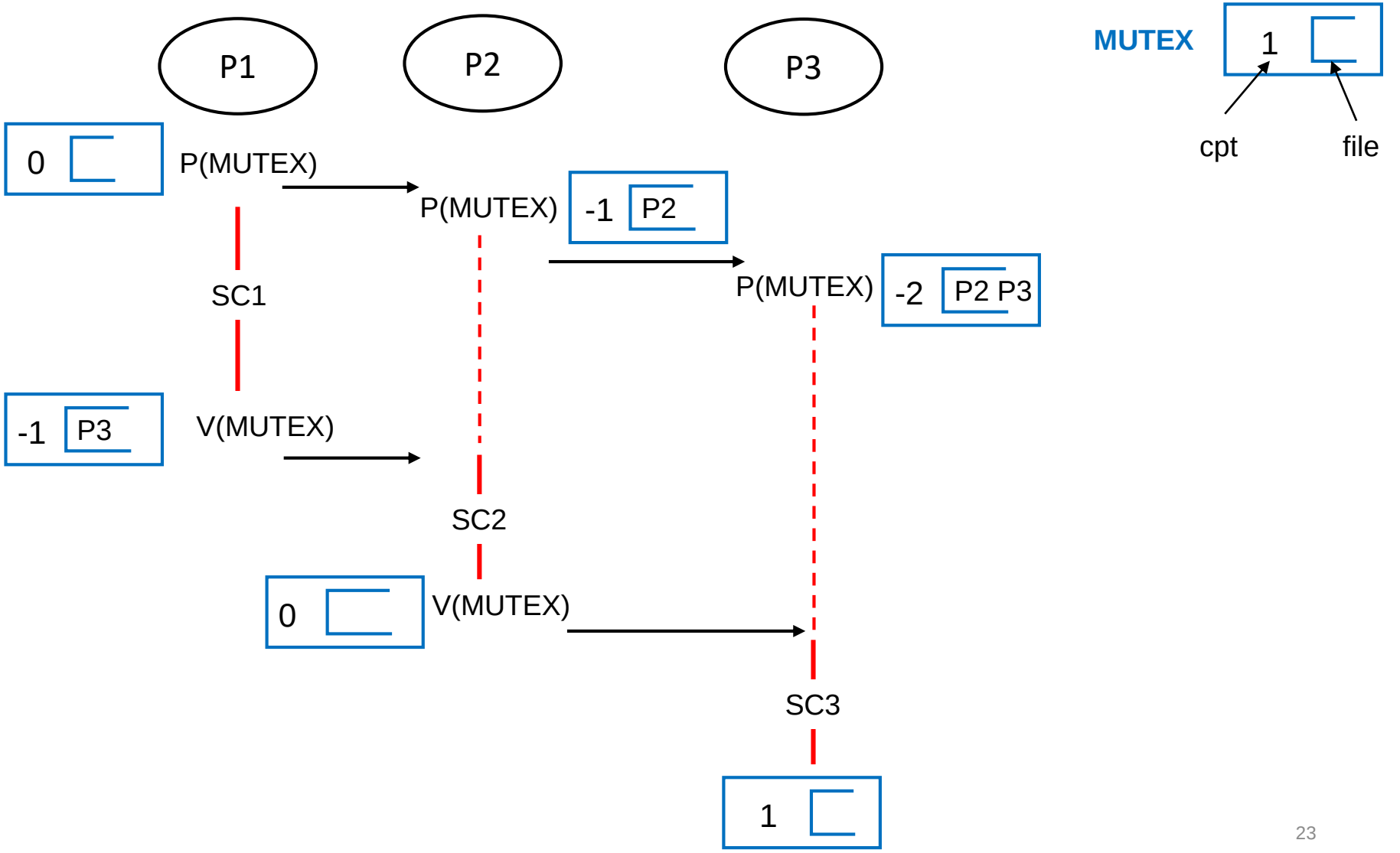
Problèmes Classiques : Exclusion mutuelle

- Autoriser un seul processus à accéder à la SC

$\Rightarrow$ Définir un sémaphore avec un compteur = 1

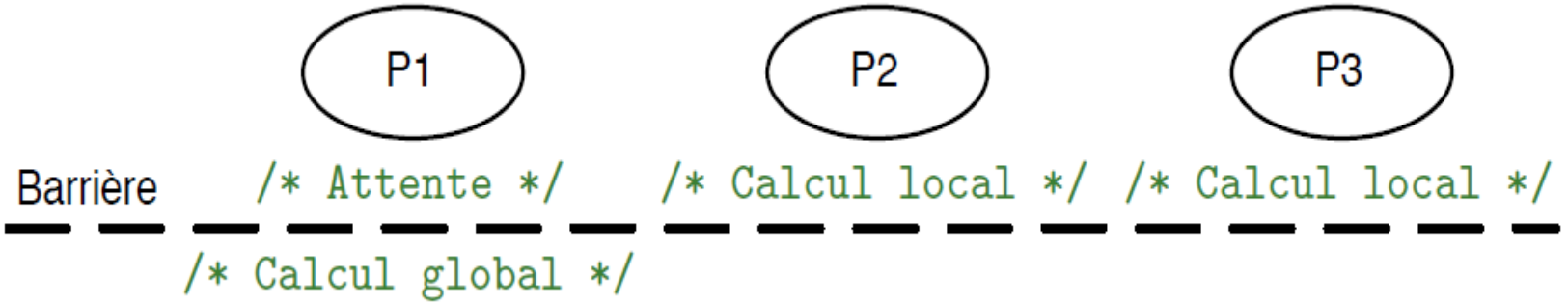
```
                    Init(MUTEX, 1);

EntrerSC        [   P(MUTEX);

                    /* Section critique */

SortirSC        [   V(MUTEX);
```

Exclusion mutuelle - Exemple



Problèmes classiques - Barrière

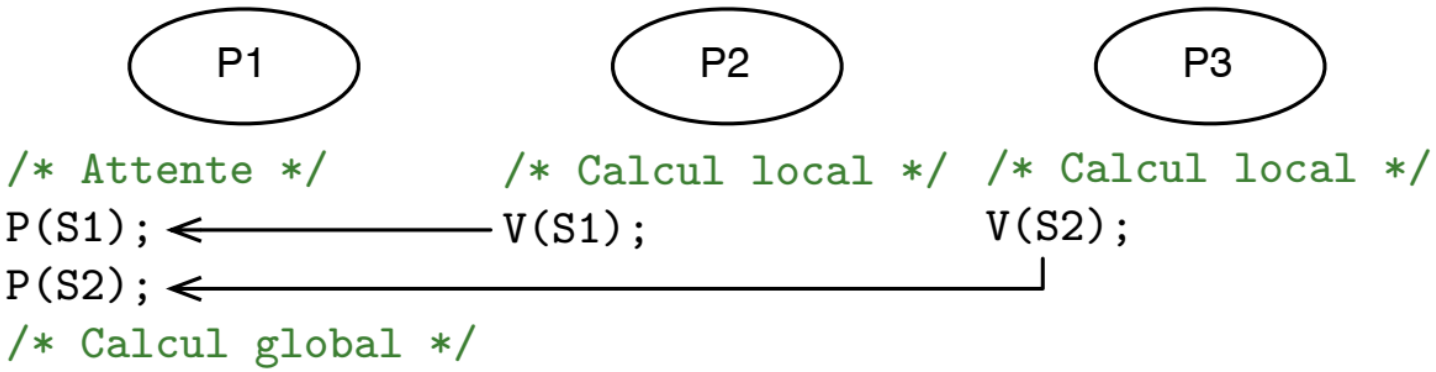
- Forcer un processus à attendre les autres



– Un sémaphore S1 pour attendre le processus P2

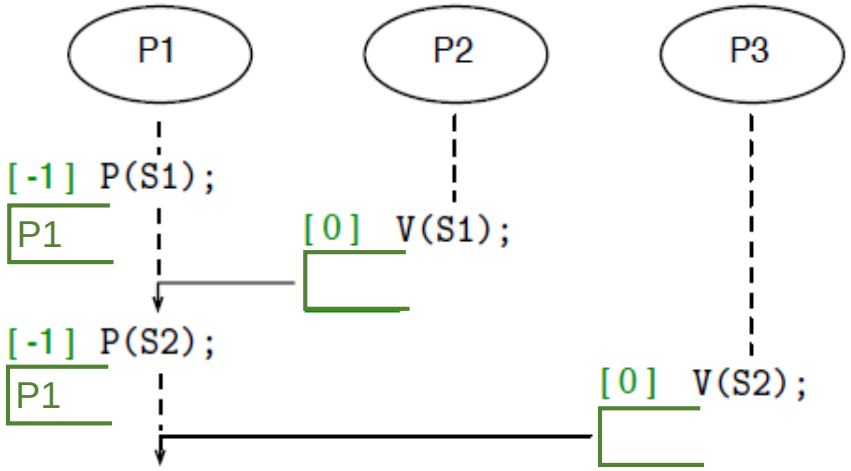
– Un sémaphore S2 pour attendre le processus P3

Problèmes classiques - Barrière

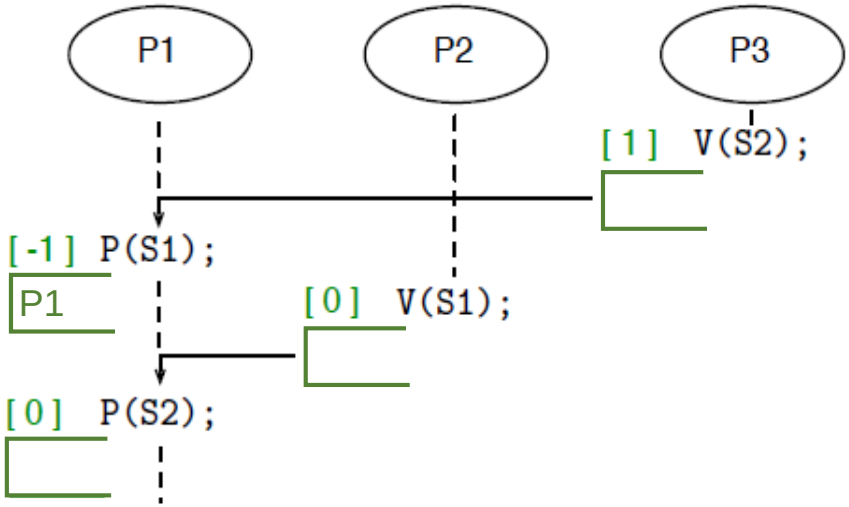


Barrière - Exemple

Scénario 1

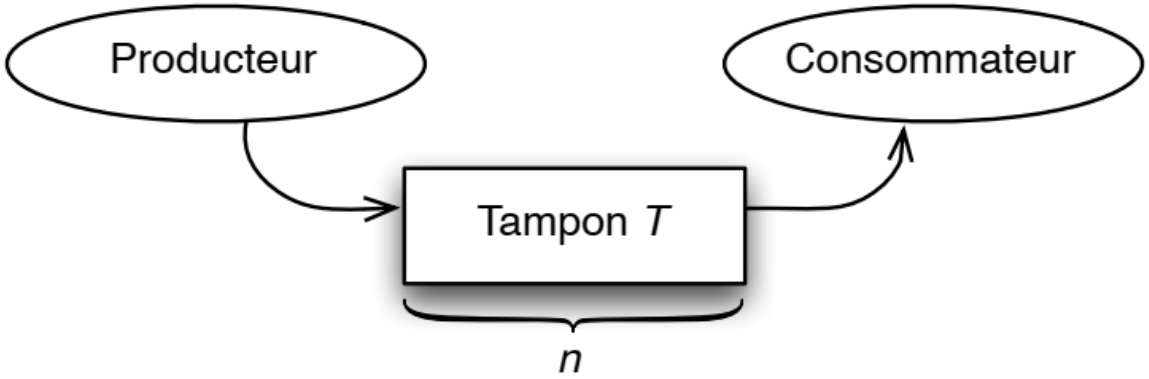


Scénario 2



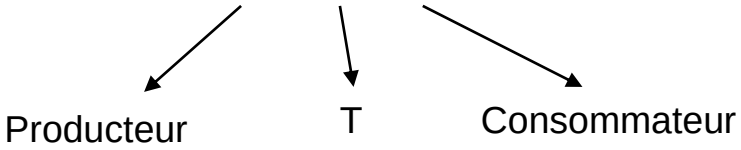
Problèmes classiques : Producteur/Consommateur

- Processus Producteur et Consommateur coopèrent en se partageant un tampon T de taille *n*.



- Le producteur **dépose** un message sur une case du tampon
- Le consommateur **retire** le message du tampon
- Pas forcement à la même vitesse

- ex : $ ls –l | wc –l



Producteur/Consommateur

- 3 contraintes
  - Le producteur ne peut pas déposer un message alors que T est **plein**
  - Le consommateur ne peut pas retirer un message alors que T est **vide**
  - Le producteur et le consommateur ne doivent pas accéder simultanément à la même case.

- Sémaphores pour résoudre les contraintes
  - *Scons*: pour bloquer les consommateurs
    - Initialement 0 consommation autorisée

    => Initialisé à 0 **Init(Scons,0)**
  - *Sprod* : pour bloquer les producteurs
    - Initialement *n* productions autorisées

    => Initialisé à *n* **Init(Sprod,n)**
  - *Mutex* :
    - Initialisé à 1 **Init(Mutex,1)**
    - Exclusion mutuelle pour l'accès à chaque case de T

Producteur/Consommateur

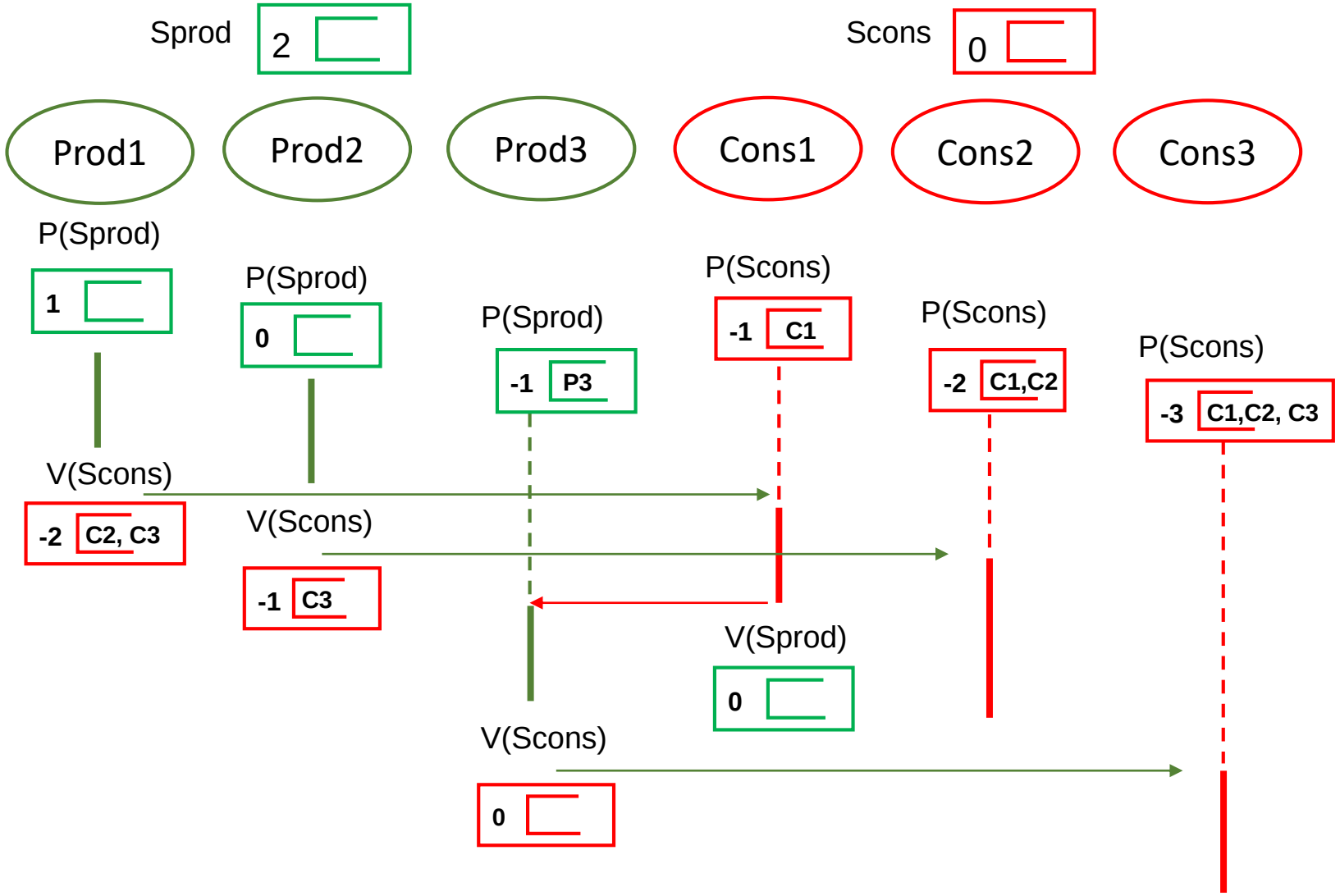
PRODUCTEUR

```
while (true)  {
    / * produire un  message */
    P (sprod);

    P (Mutex);
     /* déposer le message dans T */
    V (Mutex);

    V (Scons);
 }
```

CONSOMMATEUR

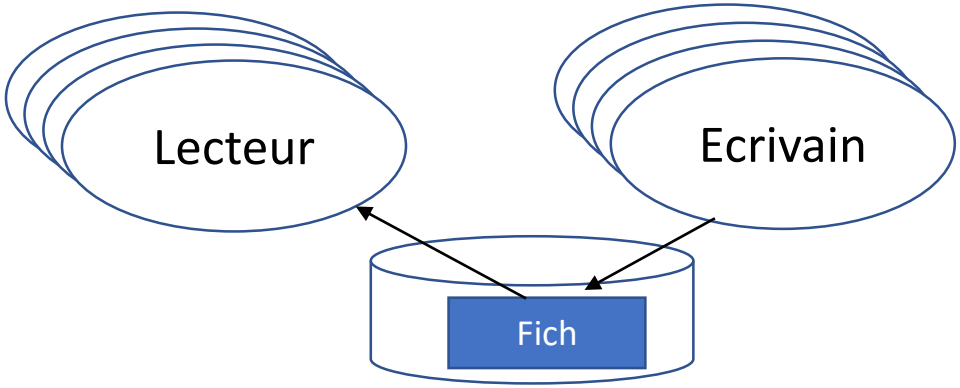
```
while (true) {
   P (Scons);

     P (Mutex);
   /* retirer un message de T */
   V (Vutex);

   V (Sprod);

   /* consommer le message */
}
```

Producteur/Consommateur – Exemple n=2



Problèmes classiques : Lecteur / Ecrivain

- Des processus partagent un fichier



Contraintes de cohérence

1 seul écrivain
⊕ (XOR)
Plusieurs lecteurs

Lecteur

```
OuvreLecture();
        /* Lecture */
FermeLecture();
```

Ecrivain

```
OuvreEcriture();
        /* Ecriture */
FermeEcriture();
```

Lecteur/écrivain

- Idée : Le **premier** lecteur "verrouille" le fichier (bloque l'écriture), le **dernier** lecteur "déverrouille " le fichier (débloque l'écriture)

⇒ 1 variable partagée
  - int nblect = 0;

- Sémaphores
  - MUTEX pour protéger la variable nblect : Init(MUTEX, 1);
  - Un sémaphore E pour bloquer les écrivains : Init(E, 1);

Lecteur / écrivain

```
OuvreLecture() {
    P(MUTEX);
    nblect++;
    if(nblect == 1)
        P(E);
    V(MUTEX);
}

FermeLecture() {
    P(MUTEX);
    nblect--;
    if(nblect == 0)
        V(E);
    V(MUTEX);
}
```

```
OuvreEcriture() {
    P(E);
}

FermeEcriture() {
    V(E);
}
```

Risque de famine des écrivains
(en TD solution équitable)

Interblocage

- Attente mutuelle de processus

P1 attend une ressource détenue par P2
P2 attend une ressource détenue par P3
…
PN attend une ressource détenue par P1

exemple : Init(Mutex1, 1) protège variable partagée a

Init(Mutex2, 1) protège variable partagée b