Systèmes d'exploitation

Synchronisation

Damien MASSON
http://www.esiee.fr/~massond/

Dernière modification: 12 mai 2026

Informations pratiques

- Damien Masson – Bureau 4206 – `http://www.esiee.fr/~massond`
- Les supports ne contiennent pas tout :
  - Prendre des notes
  - Poser des questions
  - Lire le poly !

Exemple de problème de concurrence

- Thread 1 :

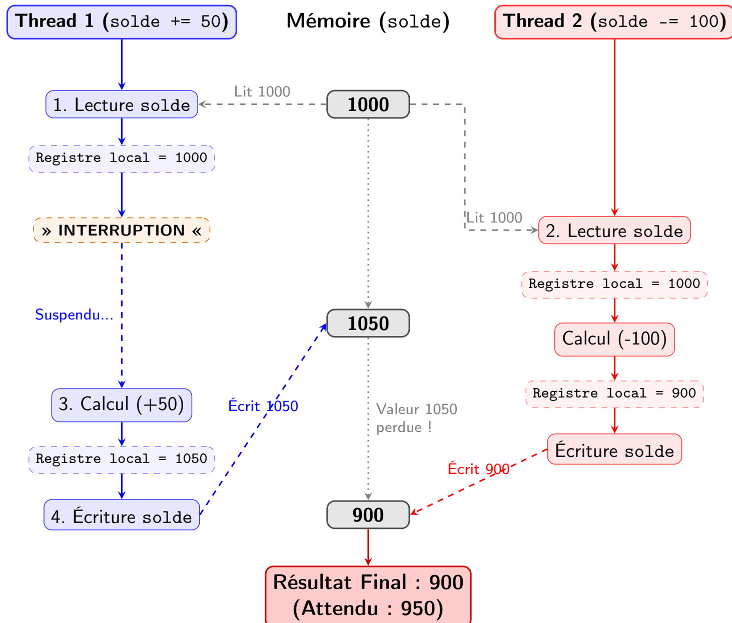
```
void credit(int montant1) {
  solde += montant1;
}
```

- Thread 2 :

```
void debit(int montant2) {
  if(solde > montant2)
    solde = solde - montant2;
}
```

- Situation :
  - `solde = 1000` ; `montant1 = 50` ; `montant2 = 100`
- Résultat attendu : `solde = 950`
- Résultat possible : `solde = 900`

Schéma : L'entrelacement fatal (Race Condition)



1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX
3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set

2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX

3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX
3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Section Critique

Dans un système multi-threads :

- **Section critique** : fragment de code, qui à un instant donné, ne peut être exécuté que par un unique thread.
- La section critique est le fragment de code qui manipule une ressource partagée (ressource critique).

Section Critique : Solution

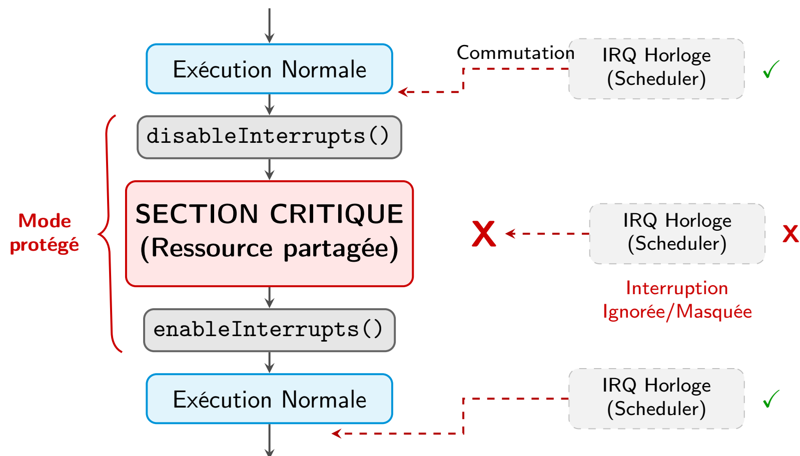
Opérations spéciales à l'entrée et à la sortie de la section critique qui respectent les conditions suivantes :

- **Exclusion mutuelle**
- Indépendance par rapport à la vitesse d'exécution des threads
- Progression
- Attente limitée

Masquage des interruptions : Principe

- Avant d'entrer en section critique : masquage des interruptions (*disable interrupts*)
- Avant de sortir de section critique : restauration des interruptions (*enable interrupts*)
- **Objectif :** Le processus ne peut plus être suspendu par le scheduler (qui dépend de l'horloge) pendant qu'il est dans la section critique.

Schéma : Exclusion mutuelle par masquage



Masquage des interruptions : Problèmes

- **Technique dangereuse :** si le thread plante ou "oublie" de restaurer les interruptions à la sortie, le système entier est figé (plus de scheduler, plus de clavier/souris).
- **Inefficace en multiprocesseur :** le masquage n'est local qu'à un seul cœur CPU. Un thread sur un autre cœur peut entrer dans la même section critique.
- **Trop brutal :** empêche le parallélisme de sections critiques totalement distinctes (qui utilisent des ressources différentes).

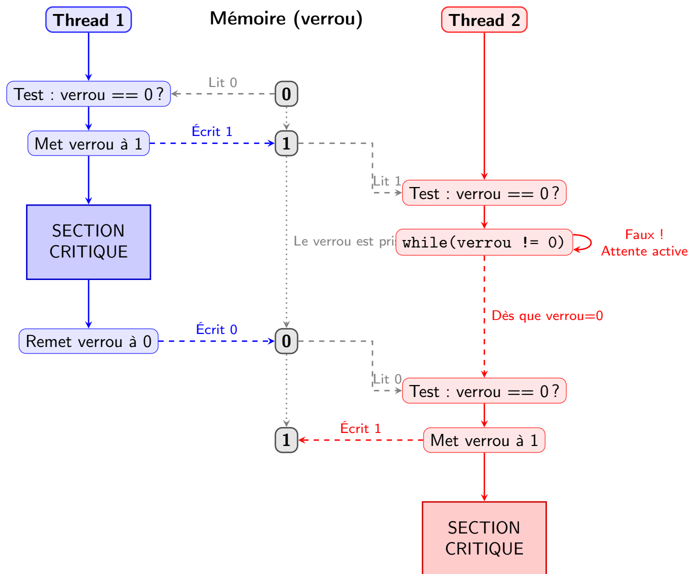
Usage

Cette technique est parfois utilisée **par le noyau du SE lui-même** pour des opérations très courtes et critiques (ex : mettre à jour la liste des processus prêts), mais elle est interdite aux processus utilisateurs.

Verrou et attente active

- Une variable partagée joue le rôle de verrou (Initialisée à 0)
- Pour entrer, un processus teste la valeur du verrou :
- **Si verrou == 0** :
  - Le processus met le verrou à 1
  - Exécute sa section critique
  - Puis remet le verrou à 0
- **Si verrou == 1** : il attend qu'il passe à 0.
  - `while(verrou != 0) ;`

Schéma : Verrou et attente active



Exemple de Verrou Naïf

- Thread 1 :

```c
void credit(int montant1){
  while(verrou!=0);
  verrou = 1;
  solde += montant1;
  verrou = 0;
}
```

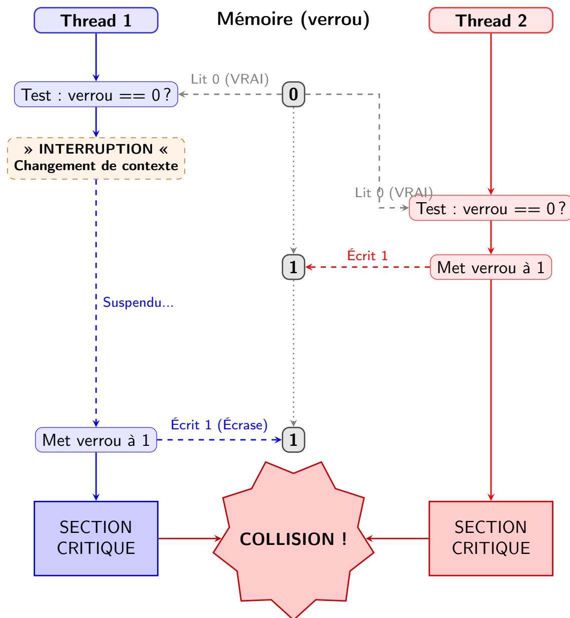
- Thread 2 :

```c
void debit(int montant2) {
  while(verrou!=0);
  verrou = 1;
  if(solde > montant2)
    solde = solde - montant2;
  verrou = 0;
}
```

Le problème demeure entier

- Verrouillage en 2 étapes : test (`while`) **puis** modification (`verrou = 1`).
- **Le problème fondamental :** Le thread peut être interrompu par le scheduler **entre** ces deux opérations !
- **Scénario d'échec :**
  1. Un Thread 1 teste le verrou, constate qu'il est à 0 (libre), et sort de la boucle `while`.
  2. **Interruption :** Le scheduler suspend le Thread 1 **avant** qu'il n'ait eu le temps de mettre le verrou à 1.
  3. Un Thread 2 s'exécute, teste le verrou (qui est toujours à 0 !), le met à 1 et entre en section critique.
  4. Le Thread 1 reprend son exécution là où il s'était arrêté : il met bêtement le verrou à 1 et entre **lui aussi** en section critique.

Schéma : Le problème du verrou naïf



Exemple de masquage

- Une solution consisterait à masquer les interruptions durant la manipulation du verrou (avec les mêmes problèmes liés à cette solution).

```
1  void entre(int verrou) {
2    disableInterrupts();
3    while(verrou){
4      enableInterrupts();
5      disableInterrupts();
6    }
7    verrou = 1;
8    enableInterrupts();
9  }
10
11 void sort(int verrou) {
12   disableInterrupts();
13   verrou = 0;
14   enableInterrupts();
15 }
```

Algorithme de Peterson naïf (Tableau d'intentions)

- **L'idée :** Pour éviter que les threads n'entrent en même temps, on utilise un tableau indiquant l'**intention** de chaque thread.
- `bool pret[2] = {false, false};`
- Un thread signale son intention (`pret[i] = true`), puis attend que l'autre ait fini (`while(pret[j])`).

Thread 0 :

```
1  pret[0] = true;
2  while (pret[1] == true); // Attente
3
4  // -- SECTION CRITIQUE --
5
6  pret[0] = false;
```

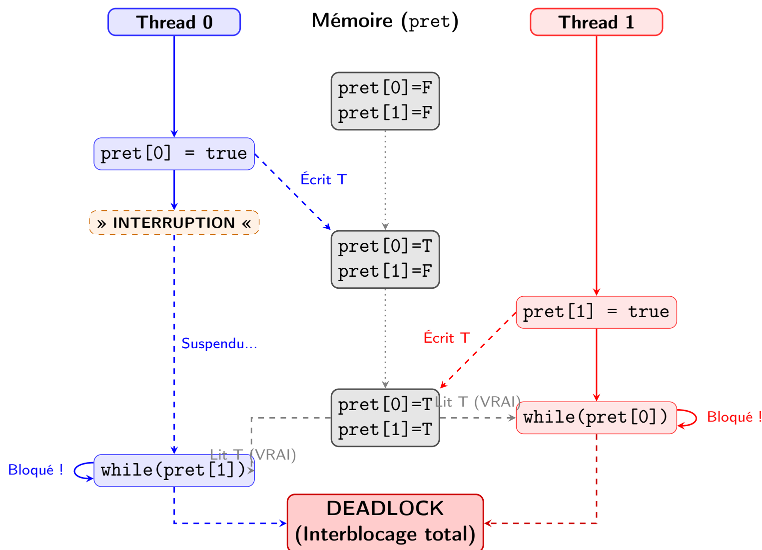
Thread 1 :

```
1  pret[1] = true;
2  while (pret[0] == true); // Attente
3
4  // -- SECTION CRITIQUE --
5
6  pret[1] = false;
```

Le problème : Risque d'interblocage (Deadlock)

- Cette approche résout bien le problème d'exclusion mutuelle (les deux threads ne peuvent pas être en section critique en même temps).
- **Cependant**, elle introduit un nouveau problème majeur : l'**interblocage** (*deadlock*).
- **Scénario fatal :**
  1. Le Thread 0 lève son drapeau (`pret[0] = true`).
  2. Le *scheduler* l'interrompt **juste avant** qu'il ne teste `pret[1]`.
  3. Le Thread 1 s'exécute, lève son drapeau (`pret[1] = true`).
  4. Le Thread 1 teste `pret[0]` : il est vrai ! Il boucle donc indéfiniment.
  5. Le Thread 0 reprend, teste `pret[1]` : il est vrai ! Il boucle aussi.
- Les deux threads s'attendent mutuellement à l'infini : le système est figé.

Schéma : L'interblocage du Peterson naïf



Attente active avec alternance (Peterson naïf)

- Solution sans intervention du système d'exploitation.
- Variable `tour` indiquant le processus qui a le droit d'entrer.

Thread 1 :

```
while (1){
  while(tour != 0) ;
  sectionCritique();
  tour = 1;
  sectionNonCritique();
}
```

Thread 2 :

```
while (1){
  while(tour != 1) ;
  sectionCritique();
  tour = 0;
  sectionNonCritique();
}
```

Problèmes de l'alternance stricte

- Hypothèse d'alternance : un thread peut être bloqué à l'entrée de la section critique même si l'autre n'est pas en section critique.
- Attente éternelle : si un des threads n'entre plus jamais en section critique, le verrou est bloqué pour l'autre.
- *La vraie solution de Peterson combine la variable ‘tour‘ et un tableau d'intention (à voir par vous-même).*

Primitives SLEEP et WAKEUP

- **SLEEP** : appel système qui suspend l'appelant en attendant qu'un autre processus le réveille.
- **WAKEUP** : réveille un thread endormi.
- Un thread qui veut entrer en section critique s'endort si un autre thread est déjà en section critique.
- Lorsque l'autre sort, il réveille celui qui s'est endormi.

Exemple SLEEP / WAKEUP

Thread 1 :

```c
while(1){
  if (cond == 0) SLEEP();
  cond = 0;
  sectionCritique();
  cond = 1;
  WAKEUP(thread2);
  sectionNonCritique();
}
```

Thread 2 :

```c
while(1){
  if (cond == 0) SLEEP();
  cond = 0;
  sectionCritique();
  cond = 1;
  WAKEUP(thread1);
  sectionNonCritique();
}
```

Critiques des solutions précédentes

- Ça ne marche toujours pas parfaitement (race condition sur le test de `cond` et l'appel à `SLEEP`).
- Difficile de généraliser à plus de deux threads.
- Implémentation délicate au niveau logiciel.
- L'attente active (quand utilisée) surcharge inutilement le CPU.

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set

2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX

3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Instruction test-and-set (Hardware)

Signature

```c
int TAS(int * v);
```

Opération **atomique** qui :

- Remplace la valeur de v par la valeur 1
- Retourne ce que valait v avant l'étape 1

Rappel

Une opération atomique est un ensemble d'instructions pendant lesquelles on a la garantie qu'aucune interruption ne se produit. Elle est implémentée matériellement (hard).

Section critique avec TAS

Valeur initiale de `verrou` : 0

```
// Attente active sécurisée grâce au matériel
while(TAS(&verrou)) ;

sectionCritique();

verrou = 0;
```

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set

2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX

3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX
3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Sémaphores, Dijkstra – 1965

- **Objectif :** autoriser un nombre de threads fixé v à être en même temps en section critique.
- Mécanisme de synchronisation géré par l'OS.
- Manipulation d'un objet entier via deux opérations atomiques :
  - **DOWN** (aussi appelée *wait* ou P)
  - **UP** (aussi appelée *post* ou V)

Principe de fonctionnement

DOWN(s) :

- Avant d'entrer, le thread invoque DOWN.
- Si $v > 0$: Décrémente v (prend un jeton).
- Si $v == 0$: Bloque le thread jusqu'à réveil.

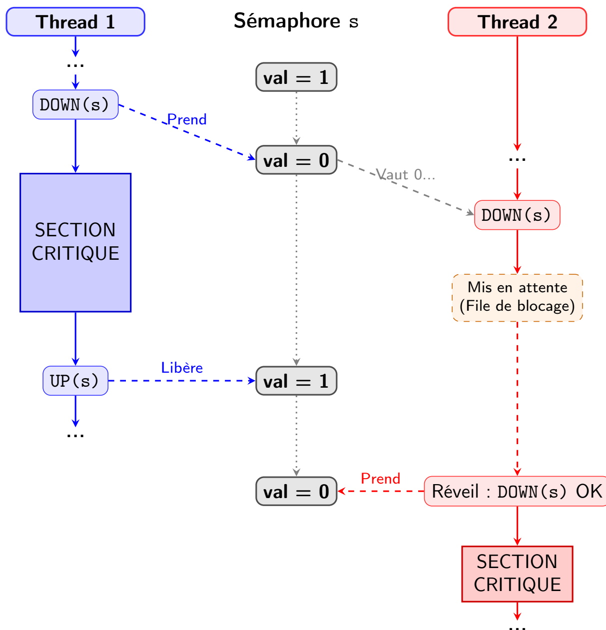
UP(s) :

- En sortant, le thread invoque UP.
- Si aucun thread n'est bloqué, incrémente v.
- Si un thread est bloqué, réveille un thread (la valeur de v reste à 0 car le jeton est directement passé).

Exemple : exclusion mutuelle (Mutex)

```
mutex = new_semaphore(1); // 1 seule place

while(1) {
  DOWN(&mutex);
  sectionCritique();
  UP(&mutex);
  sectionNonCritique();
}
```

Schéma : Exclusion mutuelle avec Sémaphore



Exemple : Séquencement (Le Problème)

- **Objectif :** Forcer un ordre d'exécution précis entre deux processus ou threads.
- **Contrainte :** L'instruction **S1** du Thread 1 **doit impérativement** s'exécuter avant l'instruction **S2** du Thread 2.
- **Le problème :** Sans synchronisation, c'est le *scheduler* qui décide. Rien ne garantit cet ordre. Si le Thread 2 est planifié en premier, S2 s'exécutera avant S1 !

Thread 1 :

```
...
// Action préalable

S1; // Doit passer en
    premier

...
```

Thread 2 :

```
...
// Doit attendre que S1
// soit terminée !
S2;

...
```

Exemple : Séquencement (La Solution)

- **Solution** : Utiliser un sémaphore `s` initialisé à **0**.
- **Principe** :
  - Le Thread 1 exécute S1, puis signale qu'il a terminé avec `UP(s)`.
  - Le Thread 2 se met en attente avec `DOWN(s)` juste avant S2.
  - Si T2 arrive trop tôt, la valeur 0 le bloque jusqu'à ce que T1 fasse son `UP`.

Initialisation : `Semaphore s = 0;`

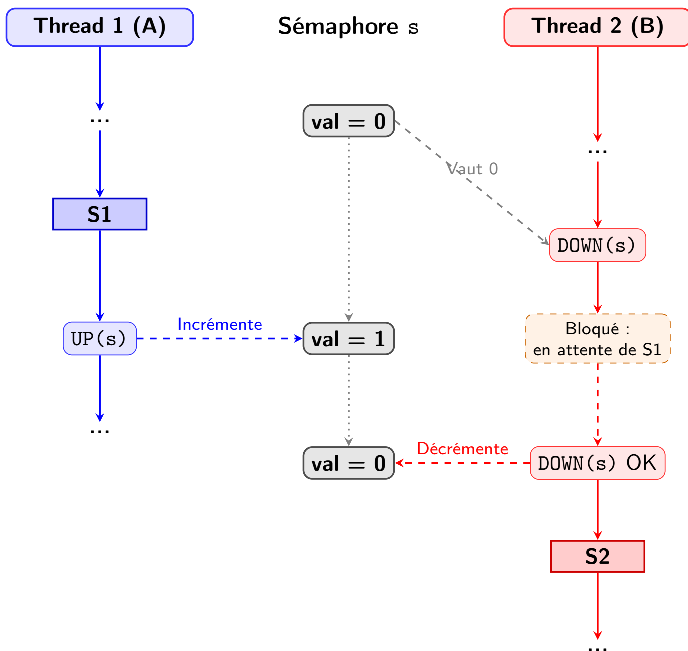
Thread 1 :

```
...

S1;
UP(s); // Donne le signal

...
```

Thread 2 :

```
...

DOWN(s); // Attend le signal
S2;

...
```

Schéma : Le Rendez-vous (Séquencement)



Implémentation OS : L'opération `wait` (DOWN)

- Un sémaphore contient une valeur, une file d'attente, et un **verrou matériel** (`lock`) pour protéger sa propre structure.
- On utilise `TAS` (Test-And-Set) pour créer un *spinlock* très court lors de la modification du sémaphore.

```
typedef struct {
    int val;
    Queue q;
    int lock; // Verrou materiel (0 = libre, 1 = pris)
} Semaphore;

void wait(Semaphore *S) {
    while (TAS(&S->lock)); // Attente active tres courte

    S->val--;
    if (S->val < 0) {
        ajouter_processus(S->q, processus_courant);
        S->lock = 0;        // IMPORTANT: liberer avant de
            dormir !
        bloquer();          // Le scheduler endort le thread
    } else {
        S->lock = 0;        // Liberer le verrou
    }
}
```

Implémentation OS : L'opération post (UP)

- Lors d'un `post`, il faut réveiller un processus si la file d'attente n'est pas vide.
- Le *spinlock* (`TAS`) garantit que deux threads ne peuvent pas modifier la file d'attente ou la valeur en même temps.

```
1  void post(Semaphore *S) {
2      while (TAS(&S->lock)); // Attente active tres courte
3      S->val++;
4      if (S->val <= 0) {// Il y a des processus en attente
5          Processus p = retirer_processus(S->q);
6          reveiller(p);// Le scheduler replace P dans la file
               des prets
7      }
8      S->lock = 0;// Liberer le verrou
9  }
```

Pourquoi garder TAS ici ?

L'attente active du `while(TAS(...))` ne dure que les quelques instructions nécessaires pour modifier `val` et gérer la file q. Contrairement à un spinlock sur une longue section critique utilisateur, cette attente est négligeable et justifiée !

Animation : Synchronisation fine DOWN / UP

Mémoire :
Sémaphore S

`val`	1
`q`	vide
`lock`	0

Thread 1

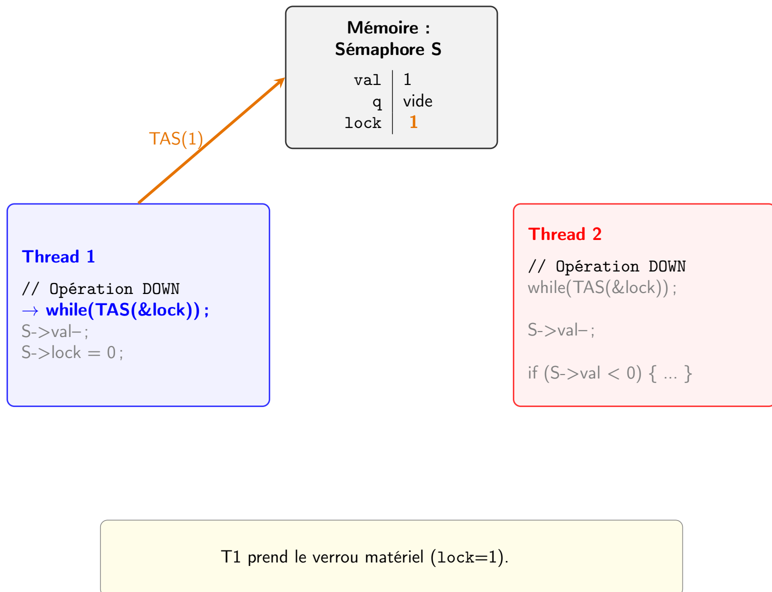
```
// Opération DOWN
while(TAS(&lock));
S->val–;
S->lock = 0;
```

Thread 2

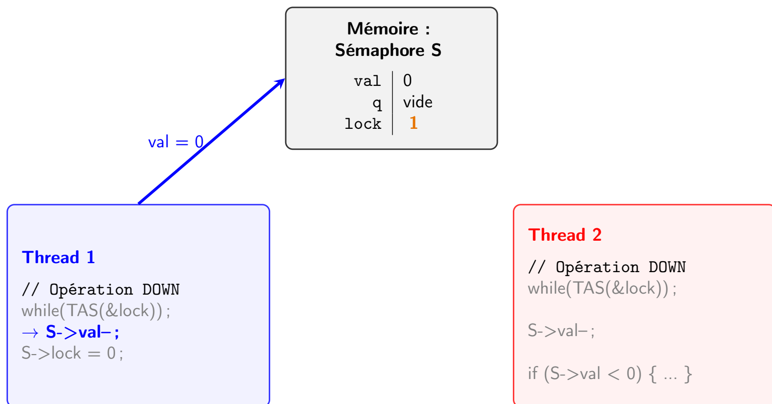
```
// Opération DOWN
while(TAS(&lock));

S->val–;

if (S->val < 0) { ... }
```

État initial : `val`=1, `lock`=0.

Animation : Synchronisation fine DOWN / UP

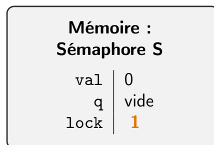


Animation : Synchronisation fine DOWN / UP



T1 décrémente `val` (0). T2 n'est pas encore arrivé.

Animation : Synchronisation fine DOWN / UP



échec (1)

Thread 1

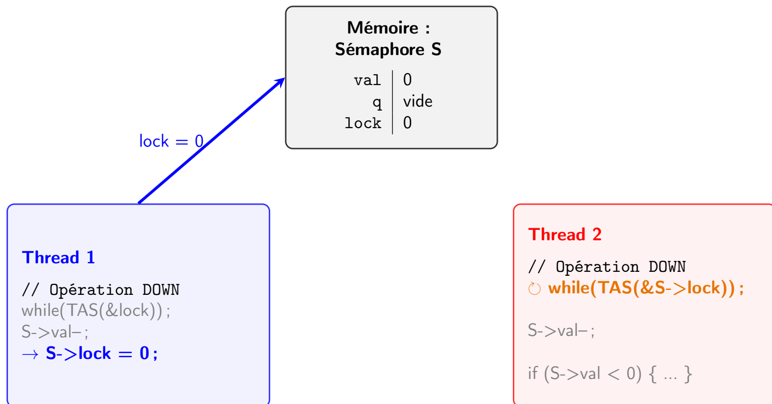
```
// Opération DOWN
while(TAS(&lock));
S->val--;
S->lock = 0;
```

Thread 2

```
// Opération DOWN
while(TAS(&S->lock));

S->val--;

if (S->val < 0) { ... }
```

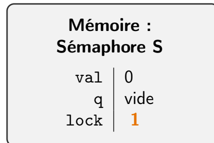
T2 arrive et tente le TAS : il échoue car T1 tient encore le `lock`.

Animation : Synchronisation fine DOWN / UP

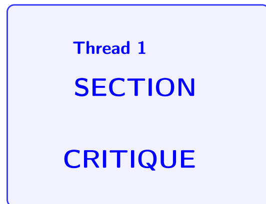


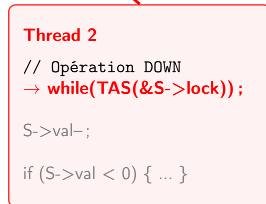
T1 a fini sa modif interne et relâche le verrou matériel (`lock`=0).

Animation : Synchronisation fine DOWN / UP



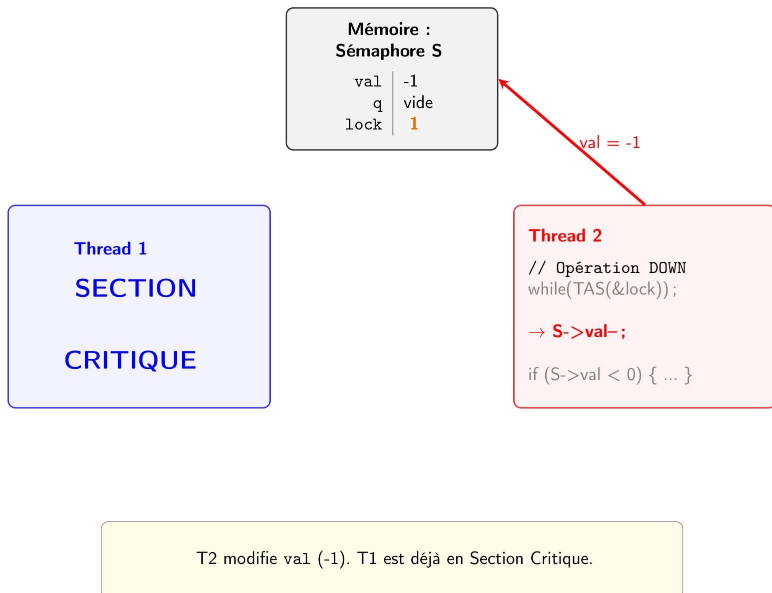
succès (1)



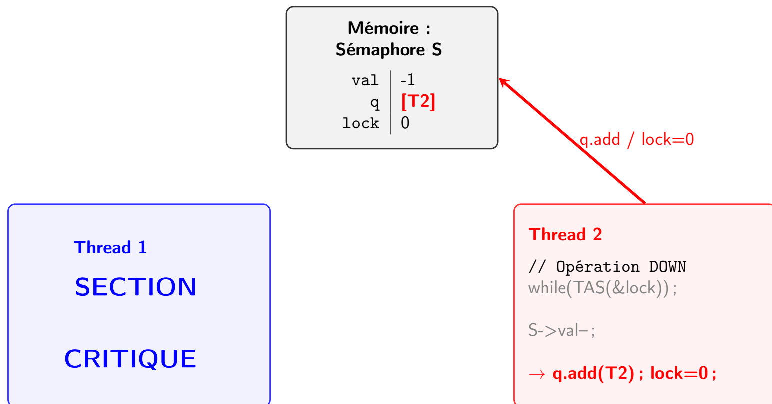


T2 boucle et retente son TAS : cette fois il réussit ! (`lock=1`).

Animation : Synchronisation fine DOWN / UP

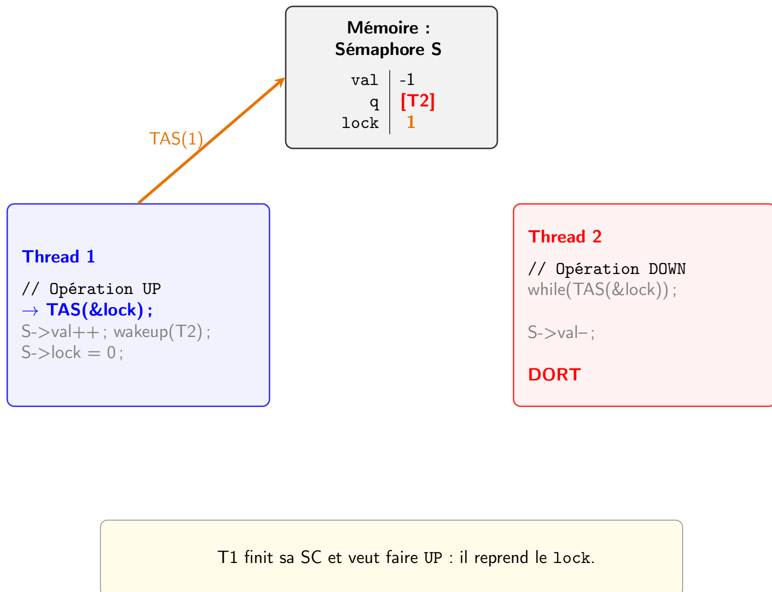


Animation : Synchronisation fine DOWN / UP

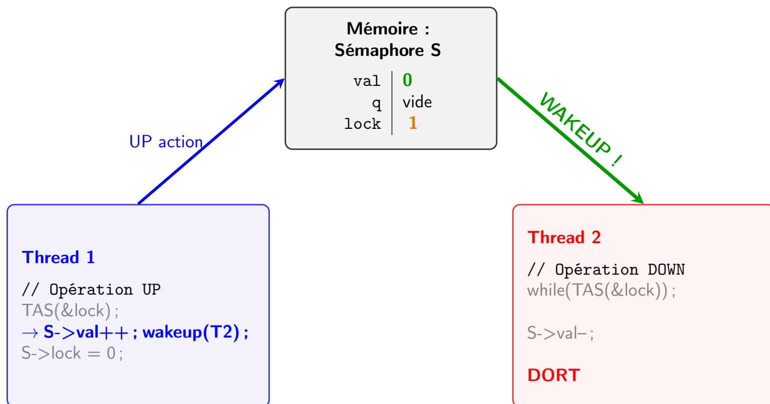


T2 voit `val` < 0 : il s'ajoute à `q`, relâche le `lock` et s'endort.

Animation : Synchronisation fine DOWN / UP

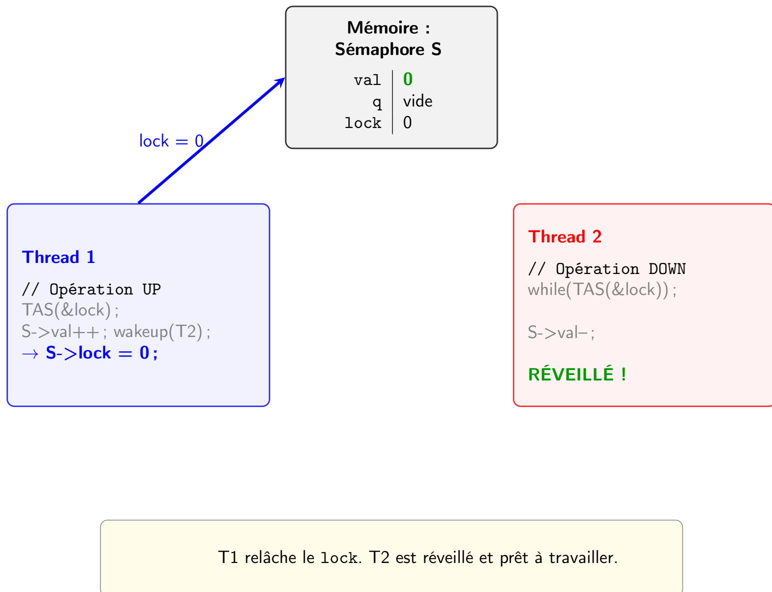


Animation : Synchronisation fine DOWN / UP



T1 incrémente `val` (0) et réveille T2 via la file q.

Animation : Synchronisation fine DOWN / UP



1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set

2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX

3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Sémaphores POSIX en C

- **Bibliothèque** : `#include <semaphore.h>`
- **Type** : `sem_t`
- **Manipulation** :
  - `sem_init(sem_t *sem, int pshared, unsigned int value)`
  - `sem_destroy(sem_t *sem)`
  - `sem_wait(sem_t *sem)` (Équivalent de DOWN)
  - `sem_post(sem_t *sem)` (Équivalent de UP)

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set

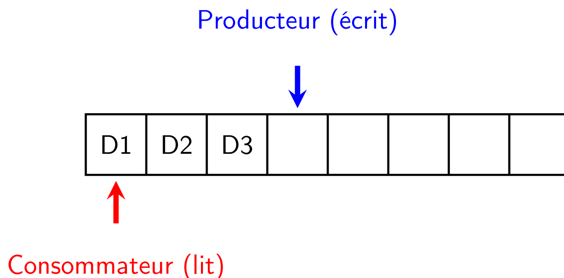
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX

3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX
3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Producteurs / Consommateurs

- Deux threads partagent un buffer circulaire de taille fixe.
- **Producteur** : ajoute des informations (Bloqué si plein).
- **Consommateur** : retire des informations (Bloqué si vide).
- Accès exclusif requis sur le buffer partagé.



Le problème des Producteurs / Consommateurs

Objectif : Faire coopérer des threads qui produisent des données et des threads qui les traitent, via un espace mémoire fini (un "buffer").

Structures de données partagées :

- Un tableau de taille fixe `N` : `type tab[N];`
- Une variable pour compter les éléments : `int n = 0;`

Variables (indices de parcours) :

- Indice d'écriture (Producteur) : `int in = 0;`
- Indice de lecture (Consommateur) : `int out = 0;`

`tab[N]` : `[ ][ ][ ][ ][ ]`

Solution naïve : Sans aucune synchronisation

Voici l'implémentation la plus basique des deux fonctions.

Producteur

```
1 void prod(type item) {
2     // Ajout
3     tab[in] = item;
4     in = (in + 1) % N;
5
6     // Mise a jour
          compteur
7     n++;
8 }
```

Consommateur

```
1 type cons() {
2     // Retrait
3     type item = tab[out];
4     out = (out + 1) % N;
5
6     // Mise a jour
          compteur
7     n--;
8     return item;
9 }
```

Animation 1 : L'entrelacement sur la variable n

Mémoire Partagée

n | 1

Producteur (n++)

1. r1 = n ;
2. r1 = r1 + 1 ;
3. n = r1 ;

Consommateur (n–)

1. r2 = n ;
2. r2 = r2 - 1 ;
3. n = r2 ;

r1: 1

r2: ?

État initial : n=1.

Animation 1 : L'entrelacement sur la variable n

Mémoire Partagée

n | 1

Producteur (n++)

1. r1 = n ; ←
2. r1 = r1 + 1 ;
3. n = r1 ;

Consommateur (n–)

1. r2 = n ;
2. r2 = r2 - 1 ;
3. n = r2 ;

`r1: 1`

`r2: ?`

Le Producteur lit `n` dans son registre `r1` (1).

Animation 1 : L'entrelacement sur la variable `n`

Mémoire Partagée

n	1

Producteur (n++)

1. r1 = n ;
2. r1 = r1 + 1 ; ←
3. n = r1 ;

Consommateur (n–)

1. r2 = n ;
2. r2 = r2 - 1 ;
3. n = r2 ;

`r1: 2`

`r2: ?`

Le Producteur calcule `r1+1` = 2. **L'OS l'interrompt ici !**

Animation 1 : L'entrelacement sur la variable `n`

Mémoire Partagée

`n` | **0**

Producteur (n++)

1. r1 = n ;
2. r1 = r1 + 1 ;
3. n = r1 ;

Consommateur (n–)

1. r2 = n ; ←
2. r2 = r2 - 1 ; ←
3. n = r2 ; ←

`r1: 2`

`r2: 0`

Le Consommateur s'exécute : il lit `n=1`, calcule `1-1=0` et **écrit n=0**.

Animation 1 : L'entrelacement sur la variable n

Mémoire Partagée

n | **2**

Producteur (n++)

1. r1 = n ;
2. r1 = r1 + 1 ;
3. n = r1 ; ←

Consommateur (n–)

1. r2 = n ;
2. r2 = r2 - 1 ;
3. n = r2 ;

`r1: 2`

`r2: 0`

Le Producteur reprend : il écrit
sa valeur `r1=2` en mémoire. **Échec !**

Animation 1 : L'entrelacement sur la variable n

Mémoire Partagée

n | **2**

Producteur (n++)

1. r1 = n ;
2. r1 = r1 + 1 ;
3. n = r1 ;

Consommateur (n–)

1. r2 = n ;
2. r2 = r2 - 1 ;
3. n = r2 ;

r1: 2

r2: 0

Bilan : Le retrait a été écrasé. Il faut protéger la variable n.

Protection des données : Le Mutex

Solution : Utiliser un **Mutex** (*Mutual Exclusion*).

Un Mutex fonctionne comme un **sémaphore initialisé à 1**.

- `DOWN(m)` : Prend le jeton (passe à 0). Si déjà à 0, le thread attend.
- `UP(m)` : Rend le jeton (passe à 1) et réveille un thread en attente.

Producteur

```c
void prod(type item) {
    DOWN(m); // Verrouille

    tab[in] = item;
    in = (in + 1) % N;
    n++;

    UP(m);   // Deverrouille
}
```

Consommateur

```c
type cons() {
    DOWN(m); // Verrouille

    type item = tab[out];
    out = (out + 1) % N;
    n--;

    UP(m);   // Deverrouille
    return item;
}
```

Animation 2 : Le problème de l'état du buffer

Mémoire Partagée

m (mutex)	1
n	0
tab	[ ][ ][ ]

Producteur

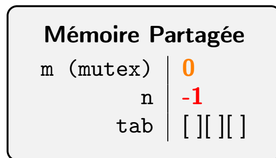
(En attente de processeur)

Consommateur

1. DOWN(m) ;
2. v=tab[out] ; n– ;
3. UP(m) ;

Le tableau est **vide** (n=0). Le Consommateur s'exécute en premier.

Animation 2 : Le problème de l'état du buffer



Producteur

(En attente de processeur)

Consommateur

1. DOWN(m) ; ←
2. v=tab[out] ; n– ;
3. UP(m) ;

Le Consommateur fait `DOWN(m)`. Le Mutex est libre, il passe.

Animation 2 : Le problème de l'état du buffer

Mémoire Partagée

`m (mutex)`	**0**
`n`	**-1**
`tab`	[ ][ ][ ]

Producteur

(En attente de processeur)

Consommateur

1. DOWN(m) ;
2. v=tab[out] ; n-- ; ←
3. UP(m) ;

Il lit une "donnée" dans le tableau vide
et décrémente `n` (**qui passe à -1 !**).

Animation 2 : Le problème de l'état du buffer

Mémoire Partagée

m (mutex)	1
n	**-1**
tab	[ ][ ][ ]

Producteur

(En attente de processeur)

Consommateur

1. DOWN(m) ;
2. v=tab[out] ; n– ;
3. UP(m) ; ←

Il libère le Mutex. Le système est dans un état incohérent.

Animation 2 : Le problème de l'état du buffer

Mémoire Partagée

`m (mutex)`	1
`n`	
`tab`	

Producteur

(En attente de processeur)

Consommateur

1. DOWN(m) ;
2. v=tab[out] ; n– ;
3. UP(m) ;

Bilan : Le Consommateur ne doit **pas** consommer si le buffer est vide (et le Producteur si c'est plein). Il faut une attente !

Tentative de correction : Mutex + Attente active

Producteur

```c
void prod(type item) {
    while(n == N); // Bloque si plein

    DOWN(m);
    tab[in] = item;
    in = (in + 1) % N;
    n++;
    UP(m);
}
```

Consommateur

```c
type cons() {
    while(n == 0); // Bloque si vide

    DOWN(m);
    type item = tab[out];
    out = (out + 1) % N;
    n--;
    UP(m);
    return item;
}
```

Pourquoi ce n'est pas satisfaisant ?

- **Attente active :** Le `while` tourne à vide et gaspille 100% du CPU.
- **Exclusion inutile :** Le Mutex bloque tout le tableau. Si in $\neq$ out, le producteur et le consommateur pourraient travailler en parallèle !

La solution finale : Sémaphores de synchronisation

On remplace `n` et le Mutex par deux **sémaphores de comptage** pour endormir les processus intelligemment :

- `vide = N` (Nombre de places libres)
- `plein = 0` (Nombre de places occupées)

Producteur

```c
void prod(type item) {
    DOWN(vide); // Attente si plein

    tab[in] = item;
    in = (in + 1) % N;

    UP(plein);  // Signale 1 donnee
}
```

Consommateur

```c
type cons() {
    DOWN(plein); // Attente si vide

    type item = tab[out];
    out = (out + 1) % N;

    UP(vide);    // Signale 1 place
    return item;
}
```

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	–
`vide`	3	–

`tab` : [ ][ ][ ]

Producteur

1. DOWN(vide) ;
2. tab[in]=v ;
3. UP(plein) ;

Consommateur

1. DOWN(plein) ; ←
2. v=tab[out] ; UP(vide) ;

Le tableau est vide. Le Consommateur tente `DOWN(plein)` (valeur 0).

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	**[Cons]**
`vide`	3	–

`tab` : [ ][ ][ ]

Producteur

1. DOWN(vide) ;
2. tab[in]=v ;
3. UP(plein) ;

Consommateur

1. DOWN(plein) ;

BLOQUÉ / ENDORMI

2. v=tab[out] ; UP(vide) ;

Endormissement : L'OS suspend le Consommateur et le met dans la file d'attente. Consommation CPU = 0.

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	**[Cons]**
`vide`	2	–

`tab` : [ ][ ][ ]

Producteur

1. DOWN(vide) ; ←
2. tab[in]=v ;
3. UP(plein) ;

Consommateur

1. DOWN(plein) ;

BLOQUÉ / ENDORMI

2. v=tab[out] ; UP(vide) ;

Le Producteur arrive. `DOWN(vide)` passe de 3 à 2.

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	**[Cons]**
`vide`	2	–

`tab` : **[X][ ][ ]**

Producteur

1. DOWN(vide) ;
2. tab[in]=v ; ←
3. UP(plein) ;

Consommateur

1. DOWN(plein) ;

BLOQUÉ / ENDORMI

2. v=tab[out] ; UP(vide) ;

Il écrit la donnée en toute sécu-
rité (le Consommateur est endormi).

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	1	**[Cons]**
`vide`	2	–

`tab` : **[X][ ][ ]**

Producteur

1. DOWN(vide) ;
2. tab[in]=v ;
3. UP(plein) ; ←

Consommateur

1. DOWN(plein) ;

BLOQUÉ / ENDORMI

2. v=tab[out] ; UP(vide) ;

`UP(plein)` : Le Producteur signale la donnée. L'OS **réveille** le Consommateur !

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	–
`vide`	2	–

`tab` : **[ ][ ][ ]**

Producteur

1. DOWN(vide) ;
2. tab[in]=v ;
3. UP(plein) ;

Consommateur

1. DOWN(plein) ;
   (Réveillé)
2. v=tab[out] ; UP(vide) ;

Le Consommateur sort de son `DOWN` (le jeton est consommé).

Animation 3 : Le blocage pris en charge par l'OS

Mémoire Partagée

Sémaphore	Valeur	File (Endormis)
`plein`	0	–
`vide`	2	–

`tab` : [ ][ ][ ]

Producteur

1. DOWN(vide) ;
2. tab[in]=v ;
3. UP(plein) ;

Consommateur

1. DOWN(plein) ;
   (Réveillé)
2. v=tab[out] ; UP(vide) ; ←

Il peut maintenant retirer la donnée en toute sécurité.

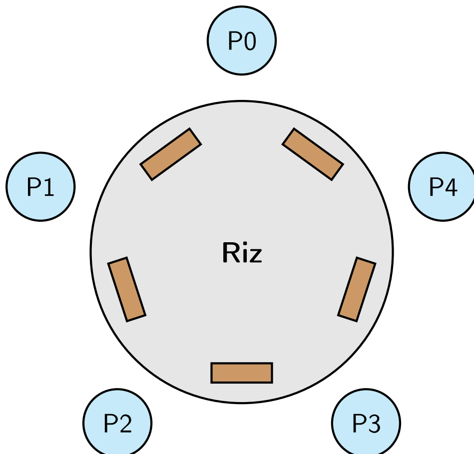
1. Section critique
- Masquage des IRQ / attente active / SLEEP et WAKEUP
- Instruction Test-and-Set

2. Sémaphores
- Principe et Implémentation
- Sémaphores POSIX

3. Problèmes
- Producteurs / Consommateurs
- Philosophes (famine et interblocage)
- Rédacteurs / Lecteurs

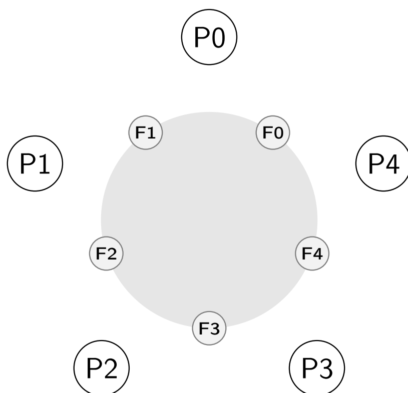
Le problème des philosophes

- 5 philosophes à table, 5 plats de riz, 5 baguettes.
- Pour manger, il faut **les 2 baguettes** adjacentes.
- Comment éviter qu'ils se bloquent tous ?



Le Dîner : 1. Solution Naïve

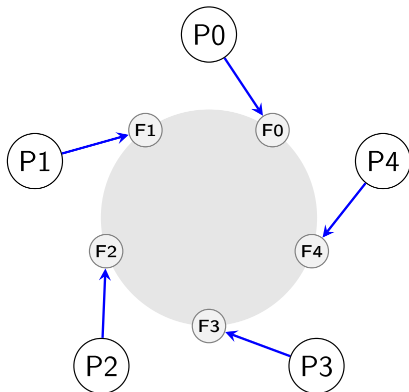
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(F[i]);
               // Gauche
6          DOWN(F[(i+1)%5]);
               // Droite
7
8          manger();
9
10         UP(F[(i+1)%5]);
11         UP(F[i]);
12     }
13 }
```



Chaque philosophe commence par prendre sa fourchette de gauche.

Le Dîner : 1. Solution Naïve

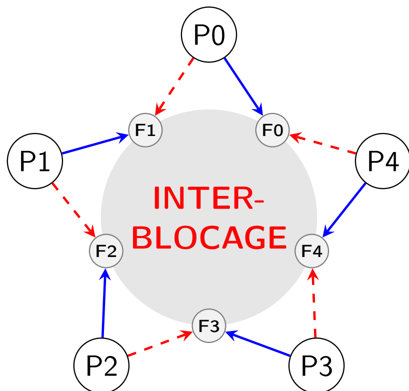
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(F[i]);
               // Gauche
6          DOWN(F[(i+1)%5]);
               // Droite
7
8          manger();
9
10         UP(F[(i+1)%5]);
11         UP(F[i]);
12     }
13 }
```



Tout le monde a faim en même temps. Chacun saisit sa fourchette gauche (`F[i]`).

Le Dîner : 1. Solution Naïve

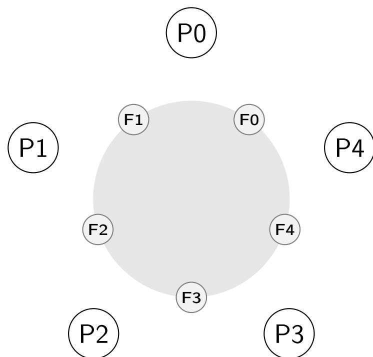
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(F[i]);
               // Gauche
6          DOWN(F[(i+1)%5]);
               // Droite
7
8          manger();
9
10         UP(F[(i+1)%5]);
11         UP(F[i]);
12     }
13 }
```



Deadlock : Chacun attend indéfiniment la fourchette droite, qui est déjà tenue par son voisin.

Le Dîner : 2. Test et Relâchement

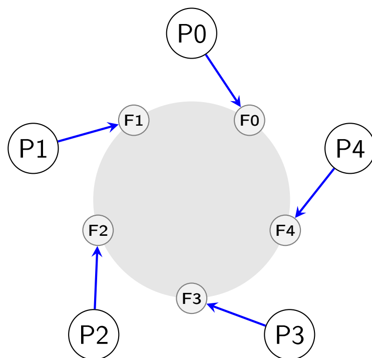
```
1  void philosophe(int i) {
2    while(1) {
3      penser();
4      DOWN(F[i]); // Gauche
5
6      // Si droite non dispo
7      if (!try_lock(F[(i+1)%5]))
         {
8        UP(F[i]); // Relache
            gauche
9        attendre_un_peu();
10       continue; // Reessaie
11     }
12
13     manger();
14     UP(F[(i+1)%5]); UP(F[i]);
15   }
16 }
```



Pour éviter l'interblocage absolu, on relâche la gauche si la droite est prise.

Le Dîner : 2. Test et Relâchement

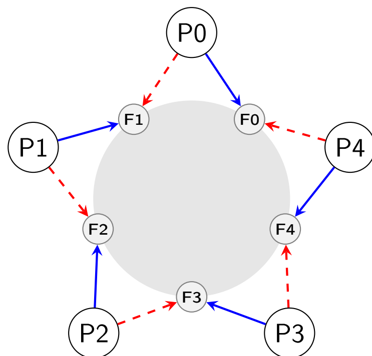
```
1  void philosophe(int i) {
2    while(1) {
3      penser();
4      DOWN(F[i]); // Gauche
5
6      // Si droite non dispo
7      if (!try_lock(F[(i+1)%5]))
         {
8        UP(F[i]); // Relache
           gauche
9        attendre_un_peu();
10       continue; // Reessaie
11     }
12
13     manger();
14     UP(F[(i+1)%5]); UP(F[i]);
15   }
16 }
```



Tout le monde prend sa fourchette gauche exactement en même temps.

Le Dîner : 2. Test et Relâchement

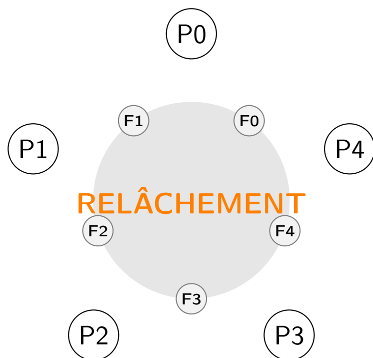
```
1  void philosophe(int i) {
2    while(1) {
3      penser();
4      DOWN(F[i]); // Gauche
5
6      // Si droite non dispo
7      if (!try_lock(F[(i+1)%5]))
           {
8        UP(F[i]); // Relache
             gauche
9        attendre_un_peu();
10       continue; // Reessaie
11     }
12
13     manger();
14     UP(F[(i+1)%5]); UP(F[i]);
15   }
16 }
```



Tout le monde vérifie la droite : elle est prise par le voisin !
(`try_lock` échoue)

Le Dîner : 2. Test et Relâchement

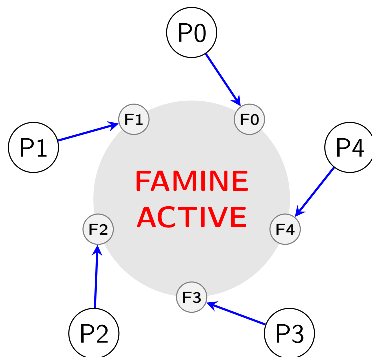
```
void philosophe(int i) {
  while(1) {
    penser();
    DOWN(F[i]); // Gauche

    // Si droite non dispo
    if (!try_lock(F[(i+1)%5]))
        {
      UP(F[i]); // Relache
          gauche
      attendre_un_peu();
      continue; // Reessaie
    }

    manger();
    UP(F[(i+1)%5]); UP(F[i]);
  }
}
```



Tout le monde relâche sa fourchette gauche par politesse.

Le Dîner : 2. Test et Relâchement

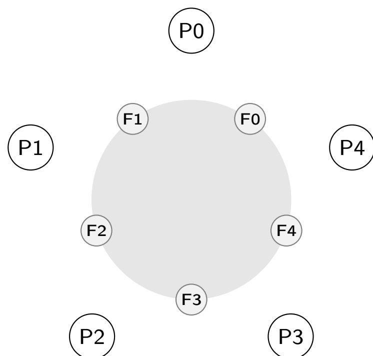
```c
void philosophe(int i) {
  while(1) {
    penser();
    DOWN(F[i]); // Gauche

    // Si droite non dispo
    if (!try_lock(F[(i+1)%5]))
        {
      UP(F[i]); // Relache
           gauche
      attendre_un_peu();
      continue; // Reessaie
    }

    manger();
    UP(F[(i+1)%5]); UP(F[i]);
  }
}
```



Livelock : Ils recommencent tous en boucle en même temps. Le CPU tourne à 100% mais personne ne mange.

Le Dîner : 3. Attente active globale

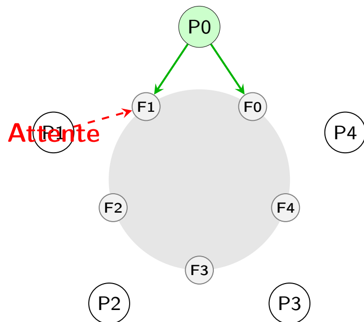
```
void philosophe(int i) {
  while(1) {
    penser();

// Attente active tant que l'
    une des deux est prise
    while(F[i] == PRISE ||
       F[(i+1)%5] == PRISE);

// Prise "atomique"
    F[i] = PRISE;
    F[(i+1)%5] = PRISE;

    manger();

    F[i] = LIBRE;
    F[(i+1)%5] = LIBRE;
  }
}
```



On tente d'éviter le deadlock en attendant que les **deux** fourchettes soient libres simultanément.

Le Dîner : 3. Attente active globale

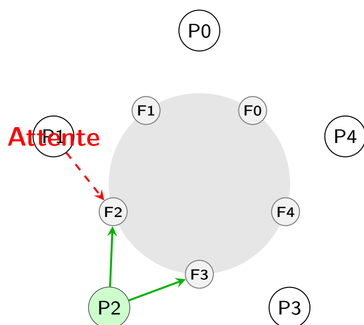
```
void philosophe(int i) {
  while(1) {
    penser();

// Attente active tant que l'
    une des deux est prise
    while(F[i] == PRISE ||
       F[(i+1)%5] == PRISE);

// Prise "atomique"
    F[i] = PRISE;
    F[(i+1)%5] = PRISE;

    manger();

    F[i] = LIBRE;
    F[(i+1)%5] = LIBRE;
  }
}
```



P0 mange (F0, F1 prises). P1 a faim mais F1 est prise, la boucle `while` de P1 tourne dans le vide.

Le Dîner : 3. Attente active globale

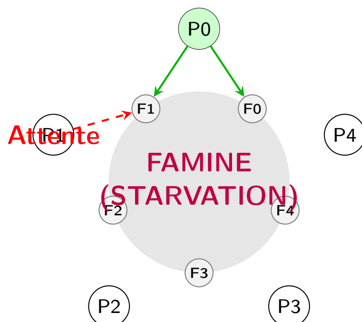
```
void philosophe(int i) {
  while(1) {
    penser();

// Attente active tant que l'
    une des deux est prise
    while(F[i] == PRISE ||
       F[(i+1)%5] == PRISE);

// Prise "atomique"
    F[i] = PRISE;
    F[(i+1)%5] = PRISE;

    manger();

    F[i] = LIBRE;
    F[(i+1)%5] = LIBRE;
  }
}
```



P0 termine, mais P2 est très rapide et prend (F2, F3). P1 voit que F2 est prise, il est toujours bloqué !

Le Dîner : 3. Attente active globale

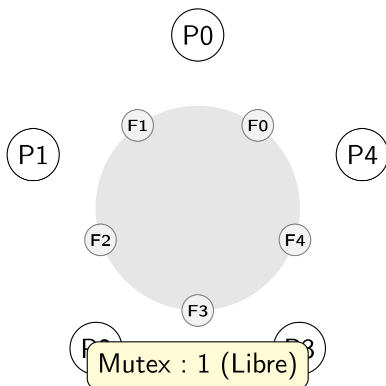
```c
void philosophe(int i) {
  while(1) {
    penser();

// Attente active tant que l'
    une des deux est prise
    while(F[i] == PRISE ||
        F[(i+1)%5] == PRISE);

// Prise "atomique"
    F[i] = PRISE;
    F[(i+1)%5] = PRISE;

    manger();

    F[i] = LIBRE;
    F[(i+1)%5] = LIBRE;
  }
}
```



Famine (Starvation) : Si P0 et P2 alternent parfaitement, P1 ne verra jamais F1 et F2 libres en même temps. Il meurt de faim alors que le système fonctionne.

Le Dîner : 3. Mutex Global

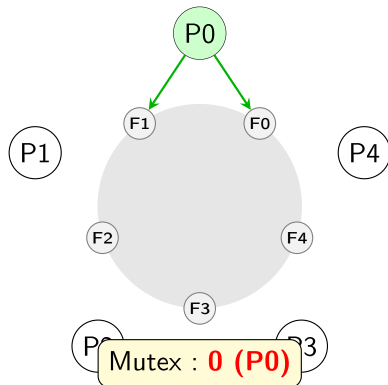
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(mutex); // Verrou table
6          DOWN(F[i]);
7          DOWN(F[(i+1)%5]);
8
9          manger();
10
11         UP(F[(i+1)%5]);
12         UP(F[i]);
13         UP(mutex);   // Libere table
14     }
15 }
```



On protège l'accès aux fourchettes par un gros Mutex global sur la table entière.

Le Dîner : 3. Mutex Global

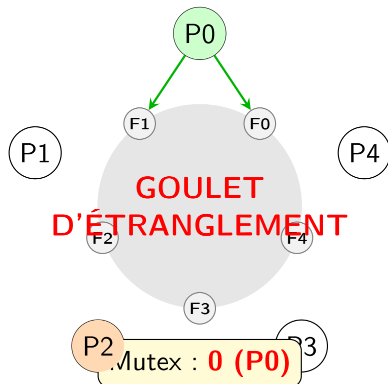
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(mutex); //
               Verrou table
6          DOWN(F[i]);
7          DOWN(F[(i+1)%5]);
8
9          manger();
10
11         UP(F[(i+1)%5]);
12         UP(F[i]);
13         UP(mutex);    //
               Libere table
14     }
15 }
```



P0 prend le Mutex, puis ses deux fourchettes, et mange paisiblement.

Le Dîner : 3. Mutex Global

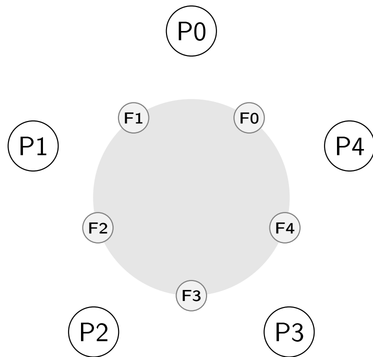
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4
5          DOWN(mutex); // Verrou table
6          DOWN(F[i]);
7          DOWN(F[(i+1)%5]);
8
9          manger();
10
11         UP(F[(i+1)%5]);
12         UP(F[i]);
13         UP(mutex);    // Libere table
14     }
15 }
```



Inefficacité : P2 a faim. Ses fourchettes (F2, F3) sont totalement libres ! Mais le Mutex est verrouillé par P0. Un seul philosophe peut manger à la fois.

Le Dîner : 4. Solution Asymétrique

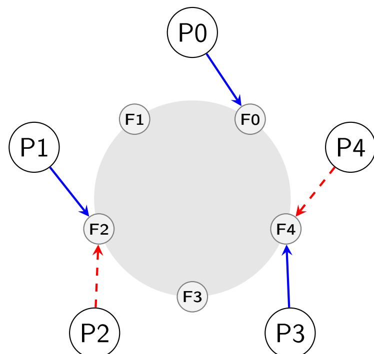
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4          if (i % 2 == 0) { // Pairs
5              DOWN(F[i]);         // Gauche
6              DOWN(F[(i+1)%5]);   // Droite
7          } else {             // Impairs
8              DOWN(F[(i+1)%5]);   // Droite
9              DOWN(F[i]);         // Gauche
10         }
11         manger();
12         UP(F[(i+1)%5]); UP(F[i]);
13     }
14 }
```



On brise la symétrie circulaire : les pairs prennent à gauche en premier, les impairs à droite.

Le Dîner : 4. Solution Asymétrique

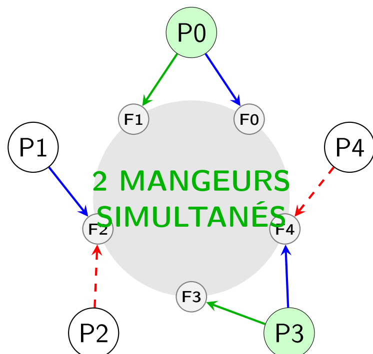
```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4          if (i % 2 == 0) { //
               Pairs
5              DOWN(F[i]);
                   // Gauche
6              DOWN(F[(i+1)%5]);
                   // Droite
7          } else {          //
               Impairs
8              DOWN(F[(i+1)%5]);
                   // Droite
9              DOWN(F[i]);
                   // Gauche
10         }
11         manger();
12         UP(F[(i+1)%5]); UP(F[i
               ]);
13     }
14 }
```



P0 prend F0. P1 prend F2. P2 veut F2 (bloqué). P3 prend F4. P4 veut F4 (bloqué).

Le Dîner : 4. Solution Asymétrique

```
1  void philosophe(int i) {
2      while(1) {
3          penser();
4          if (i % 2 == 0) { //
               Pairs
5              DOWN(F[i]);
                   // Gauche
6              DOWN(F[(i+1)%5]);
                   // Droite
7          } else {          //
               Impairs
8              DOWN(F[(i+1)%5]);
                   // Droite
9              DOWN(F[i]);
                   // Gauche
10         }
11         manger();
12         UP(F[(i+1)%5]); UP(F[i
               ]);
13     }
14 }
```



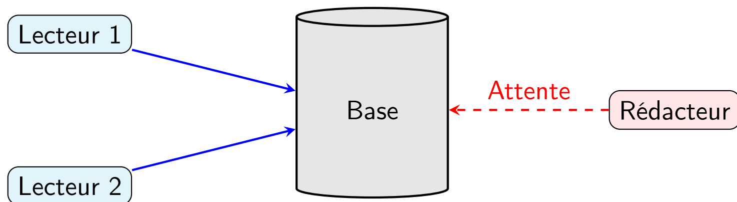
Succès : La circularité est brisée. F1 et F3 restent libres. P0 et P3 peuvent compléter leur paire et manger en parallèle !

1. Section critique
   - Masquage des IRQ / attente active / SLEEP et WAKEUP
   - Instruction Test-and-Set
2. Sémaphores
   - Principe et Implémentation
   - Sémaphores POSIX
3. Problèmes
   - Producteurs / Consommateurs
   - Philosophes (famine et interblocage)
   - Rédacteurs / Lecteurs

Rédacteurs / Lecteurs

Modélisation des accès à une base de données :

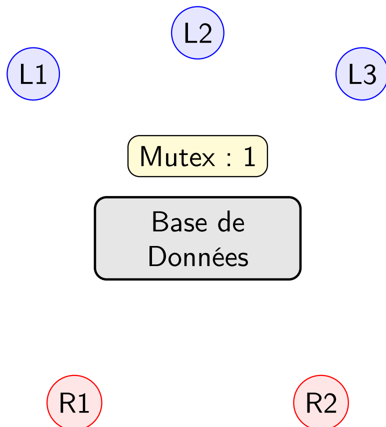
- **Cohérence :** Si un rédacteur écrit, personne d'autre n'accède.
- **Parallélisme :** Plusieurs lecteurs peuvent lire en même temps.



Lecteurs/Rédacteurs : 1. Le Mutex Unique

Lecteur / Rédacteur

```
1  void lecteur() {
2      DOWN(mutex);
3      lire_donnees();
4      UP(mutex);
5  }
6
7  void redacteur() {
8      DOWN(mutex);
9      ecrire_donnees();
10     UP(mutex);
11 }
```

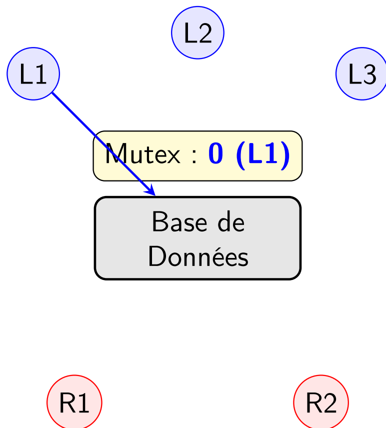


La solution de facilité : on protège toute la base de données avec un seul Mutex.

Lecteurs/Rédacteurs : 1. Le Mutex Unique

Lecteur / Rédacteur

```
1  void lecteur() {
2      DOWN(mutex);
3      lire_donnees();
4      UP(mutex);
5  }
6
7  void redacteur() {
8      DOWN(mutex);
9      ecrire_donnees();
10     UP(mutex);
11 }
```

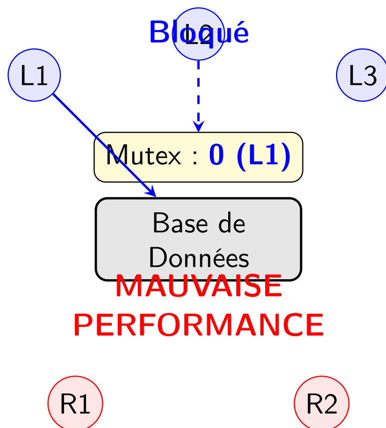


Le premier lecteur (L1) arrive, prend le Mutex et commence à lire.

Lecteurs/Rédacteurs : 1. Le Mutex Unique

Lecteur / Rédacteur

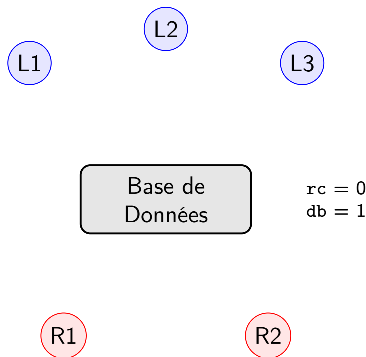
```
void lecteur() {
    DOWN(mutex);
    lire_donnees();
    UP(mutex);
}

void redacteur() {
    DOWN(mutex);
    ecrire_donnees();
    UP(mutex);
}
```



Problème : L2 veut lire. La lecture simultanée ne pose aucun risque pour les données, mais L2 est bloqué par le Mutex de L1 ! On perd l'avantage du multi-threading.

Lecteurs/Rédacteurs : 2. Priorité aux lecteurs

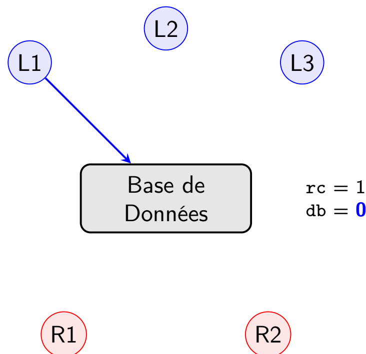
```
1  int rc = 0; // Compteur lecteurs
2
3  void lecteur() {
4      DOWN(mutex_rc);
5      rc++;
6      if (rc == 1) DOWN(db); // Le 1er
           bloque db
7      UP(mutex_rc);
8
9      lire_donnees();
10
11     DOWN(mutex_rc);
12     rc--;
13     if (rc == 0) UP(db);   // Le dernier
            libere
14     UP(mutex_rc);
15 }
16
17 void redacteur() {
18     DOWN(db);
19     ecrire_donnees();
20     UP(db);
21 }
```



Seul le premier lecteur verrouille l'accès aux rédacteurs. Les autres passent directement.

Lecteurs/Rédacteurs : 2. Priorité aux lecteurs

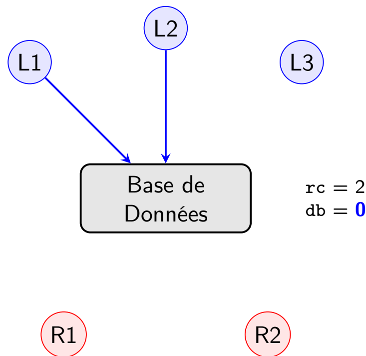
```
1  int rc = 0; // Compteur lecteurs
2
3  void lecteur() {
4      DOWN(mutex_rc);
5      rc++;
6      if (rc == 1) DOWN(db); // Le 1er
           bloque db
7      UP(mutex_rc);
8
9      lire_donnees();
10
11     DOWN(mutex_rc);
12     rc--;
13     if (rc == 0) UP(db);    // Le dernier
            libere
14     UP(mutex_rc);
15 }
16
17 void redacteur() {
18     DOWN(db);
19     ecrire_donnees();
20     UP(db);
21 }
```



L1 arrive. C'est le premier (rc=1), il verrouille le sémaphore db.

Lecteurs/Rédacteurs : 2. Priorité aux lecteurs

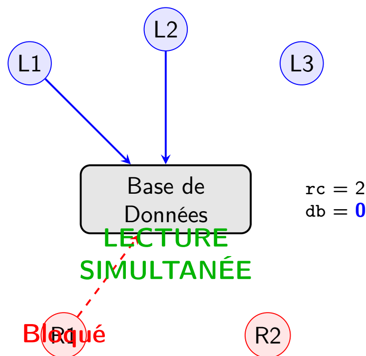
```
1  int rc = 0; // Compteur lecteurs
2
3  void lecteur() {
4      DOWN(mutex_rc);
5      rc++;
6      if (rc == 1) DOWN(db); // Le 1er
           bloque db
7      UP(mutex_rc);
8
9      lire_donnees();
10
11     DOWN(mutex_rc);
12     rc--;
13     if (rc == 0) UP(db);    // Le dernier
            libere
14     UP(mutex_rc);
15 }
16
17 void redacteur() {
18     DOWN(db);
19     ecrire_donnees();
20     UP(db);
21 }
```



L2 arrive. `rc=2`, il ne refait pas un `DOWN(db)`. **Succès : L1 et L2 lisent ensemble !**

Lecteurs/Rédacteurs : 2. Priorité aux lecteurs

```
1  int rc = 0; // Compteur lecteurs
2
3  void lecteur() {
4      DOWN(mutex_rc);
5      rc++;
6      if (rc == 1) DOWN(db); // Le 1er
           bloque db
7      UP(mutex_rc);
8
9      lire_donnees();
10
11     DOWN(mutex_rc);
12     rc--;
13     if (rc == 0) UP(db);   // Le dernier
            libere
14     UP(mutex_rc);
15 }
16
17 void redacteur() {
18     DOWN(db);
19     ecrire_donnees();
20     UP(db);
21 }
```



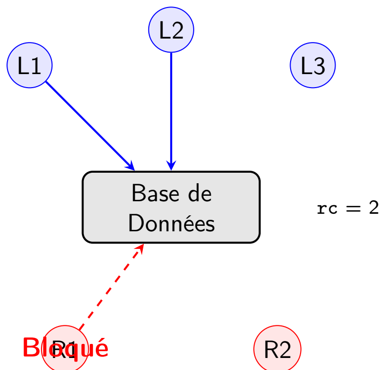
R1 veut écrire. Il fait `DOWN(db)` mais la valeur est à 0. Il est légitimement bloqué pendant les lectures.

Lecteurs/Rédacteurs : 3. La Famine des Rédacteurs

Le piège de la solution précédente :

Tant qu'il y a **au moins un lecteur** actif (`rc > 0`), le sémaphore `db` n'est jamais libéré.

Les nouveaux lecteurs contournent le verrou `db` grâce au `if (rc == 1)`.



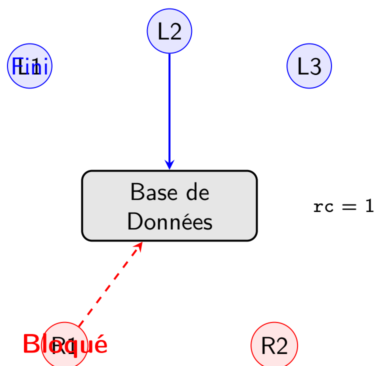
L1 et L2 lisent. R1 attend patiemment que `rc` retombe à 0.

Lecteurs/Rédacteurs : 3. La Famine des Rédacteurs

Le piège de la solution précédente :

Tant qu'il y a **au moins un lecteur** actif (`rc > 0`), le sémaphore `db` n'est jamais libéré.

Les nouveaux lecteurs contournent le verrou `db` grâce au `if (rc == 1)`.



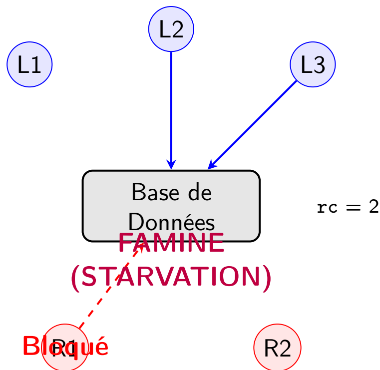
L1 a fini ! Il fait `rc-` (`rc=1`). Mais L2 lit encore, donc `db` reste verrouillé.

Lecteurs/Rédacteurs : 3. La Famine des Rédacteurs

Le piège de la solution précédente :

Tant qu'il y a **au moins un lecteur** actif (`rc > 0`), le sémaphore `db` n'est jamais libéré.

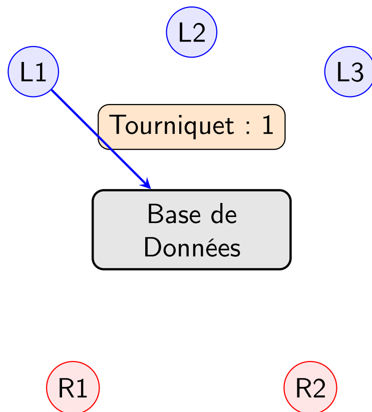
Les nouveaux lecteurs contournent le verrou `db` grâce au `if (rc == 1)`.



Famine : Un nouveau lecteur, L3, arrive (`rc=2`). Si les lecteurs s'enchaînent continuellement (L2 part, L1 revient, etc.), `rc` ne sera jamais à 0. **Le rédacteur meurt de faim.**

Lecteurs/Rédacteurs : 4. Le Tourniquet Équitable

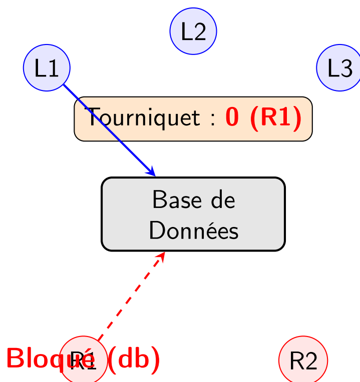
```
1  // Ajout d'un semaphore tourniquet (1)
2
3  void lecteur() {
4      DOWN(tourniquet); // File d'attente
5      DOWN(mutex_rc);
6      rc++;
7      if (rc == 1) DOWN(db);
8      UP(mutex_rc);
9      UP(tourniquet);   // Passe le
           portillon
10
11     lire_donnees();
12     // ... decrementation de rc ...
13 }
14
15 void redacteur() {
16     DOWN(tourniquet); // Prend sa place
17     DOWN(db);
18     UP(tourniquet);   // Laisse le
           suivant
19     ecrire_donnees();
20     UP(db);
21 }
```



On ajoute un "tourniquet". Tout le monde doit y passer un par un avant d'entrer en zone critique.

Lecteurs/Rédacteurs : 4. Le Tourniquet Équitable

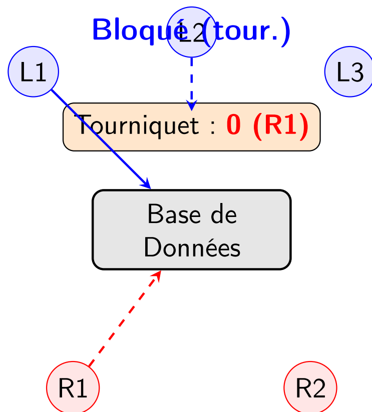
```
1   // Ajout d'un semaphore tourniquet (1)
2
3   void lecteur() {
4       DOWN(tourniquet); // File d'attente
5       DOWN(mutex_rc);
6       rc++;
7       if (rc == 1) DOWN(db);
8       UP(mutex_rc);
9       UP(tourniquet);   // Passe le
            portillon
10
11      lire_donnees();
12      // ... decrementation de rc ...
13  }
14
15  void redacteur() {
16      DOWN(tourniquet); // Prend sa place
17      DOWN(db);
18      UP(tourniquet);   // Laisse le
            suivant
19      ecrire_donnees();
20      UP(db);
21  }
```



L1 lit. R1 arrive, passe le tourniquet (`DOWN`), et attend `db`. **Mais il garde le tourniquet bloqué !**

Lecteurs/Rédacteurs : 4. Le Tourniquet Équitable

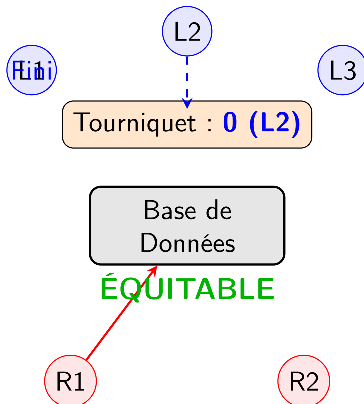
```
1  // Ajout d'un semaphore tourniquet (1)
2
3  void lecteur() {
4      DOWN(tourniquet); // File d'attente
5      DOWN(mutex_rc);
6      rc++;
7      if (rc == 1) DOWN(db);
8      UP(mutex_rc);
9      UP(tourniquet);   // Passe le
           portillon
10
11     lire_donnees();
12     // ... decrementation de rc ...
13 }
14
15 void redacteur() {
16     DOWN(tourniquet); // Prend sa place
17     DOWN(db);
18     UP(tourniquet);   // Laisse le
           suivant
19     ecrire_donnees();
20     UP(db);
21 }
```



L2 arrive pour lire. Le tourniquet est bloqué par R1. L2 ne peut pas court-circuiter R1 et grossir `rc`.

Lecteurs/Rédacteurs : 4. Le Tourniquet Équitable

```
1  // Ajout d'un semaphore tourniquet (1)
2
3  void lecteur() {
4      DOWN(tourniquet); // File d'attente
5      DOWN(mutex_rc);
6      rc++;
7      if (rc == 1) DOWN(db);
8      UP(mutex_rc);
9      UP(tourniquet);   // Passe le
           portillon
10
11     lire_donnees();
12     // ... decrementation de rc ...
13 }
14
15 void redacteur() {
16     DOWN(tourniquet); // Prend sa place
17     DOWN(db);
18     UP(tourniquet);   // Laisse le
           suivant
19     ecrire_donnees();
20     UP(db);
21 }
```



Équité respectée : L1 finit et libère db. R1 peut enfin écrire ! Les lecteurs arrivés après R1 doivent attendre qu'il ait fini.

Synchronisation : concepts clefs

- **Atomicité** : Instruction test-and-set (hardware)
- **Primitives OS** : Sémaphores, Mutex
- **Problèmes fatals** :
  - Interblocage (Deadlock)
  - Famine (Starvation)
  - Attente active (Gaspillage CPU)
- **3 problèmes génériques** :
  - Producteur / Consommateur (Régulation de flux)
  - Les philosophes (Partage de ressources multiples adjacentes)
  - Lecteur / Rédacteur (Accès concurrent vs exclusif)