Systèmes d'Exploitation : Threads et synchronisation

DIU "Enseigner l'informatique au lycée"

Aix-Marseille Université

Faculté des Sciences

Références

Sources :

- Cours L3 informatique "Systèmes d'exploitation" Leonardo Brenner - Jean-Luc Massat
- Cours L3 informatique Télé-enseignement "Systèmes d'exploitation" Jean-Marc Talbot

Table de matière

Table de matière

Processus : rappel

- unité de calcul
- ... dont les caractéristiques sont stockées dans un PCB
- ... possédant un espace mémoire propre

Processus : rappel

- unité de calcul
- ... dont les caractéristiques sont stockées dans un PCB
- ... possédant un espace mémoire propre

- nécessite des mécanismes de communications complexes (mis en place par le SE)
- commutation de contexte coûteuse

Thread - processus légers

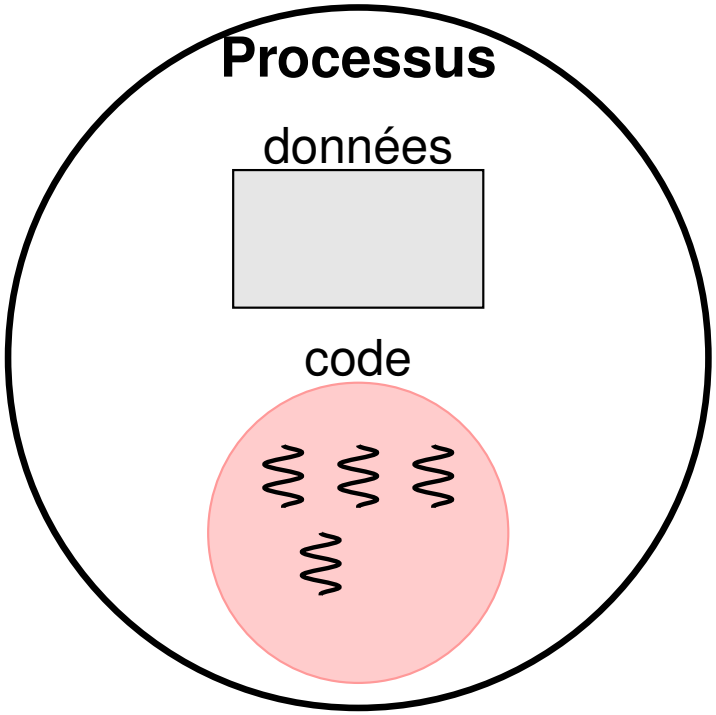
- unité de calcul (interne à un processus)
- l'espace mémoire est partagé entre les différents threads d'un même processus

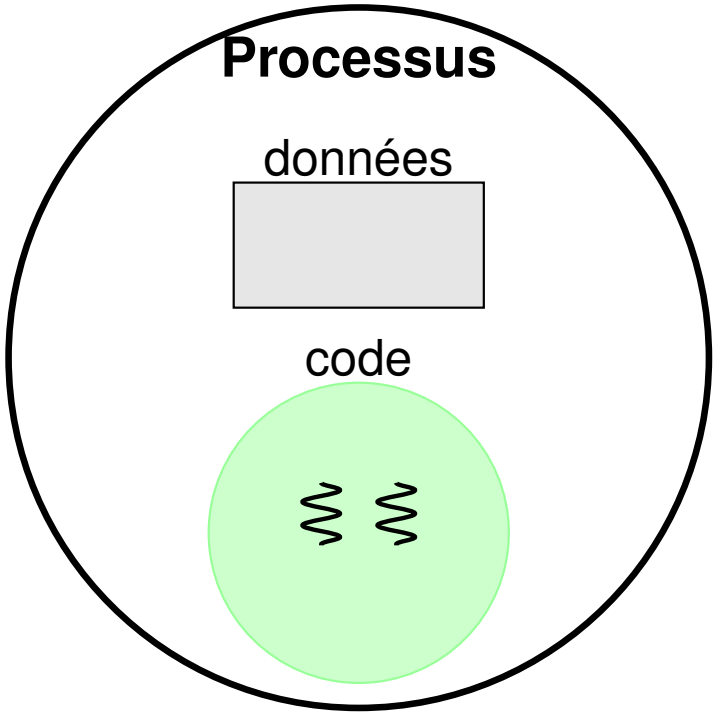
Thread - processus légers

- unité de calcul (interne à un processus)
- l'espace mémoire est partagé entre les différents threads d'un même processus

- mécanisme simple de passage entre threads
- le partage mémoire peut être source de problème

Threads et processus





Threads en python

Se trouve dans le module `threading`

```
from threading import Thread
```

Les threads définis héritent de la classe `Thread`

```
class MonThread(Thread)
```

La classe `Thread` demande l'implémentation d'une méthode `run()` qui contient le code du thread.

Threads en python

Se trouve dans le module `threading`

```
from threading import Thread
```

Les threads définis héritent de la classe `Thread`

```
class MonThread(Thread)
```

La classe `Thread` demande l'implémentation d'une méthode `run()` qui contient le code du thread.

- on utilise le constructeur pour créer un nouveau thread

  ```
  mon_thread = MonThread()
  ```
- on lance le thread qui exécute alors le code de la méthode `run()` (en parallèle du thread "lanceur")

  ```
  mon_thread.start()
  ```
- on attend la fin du thread pour poursuivre son exécution

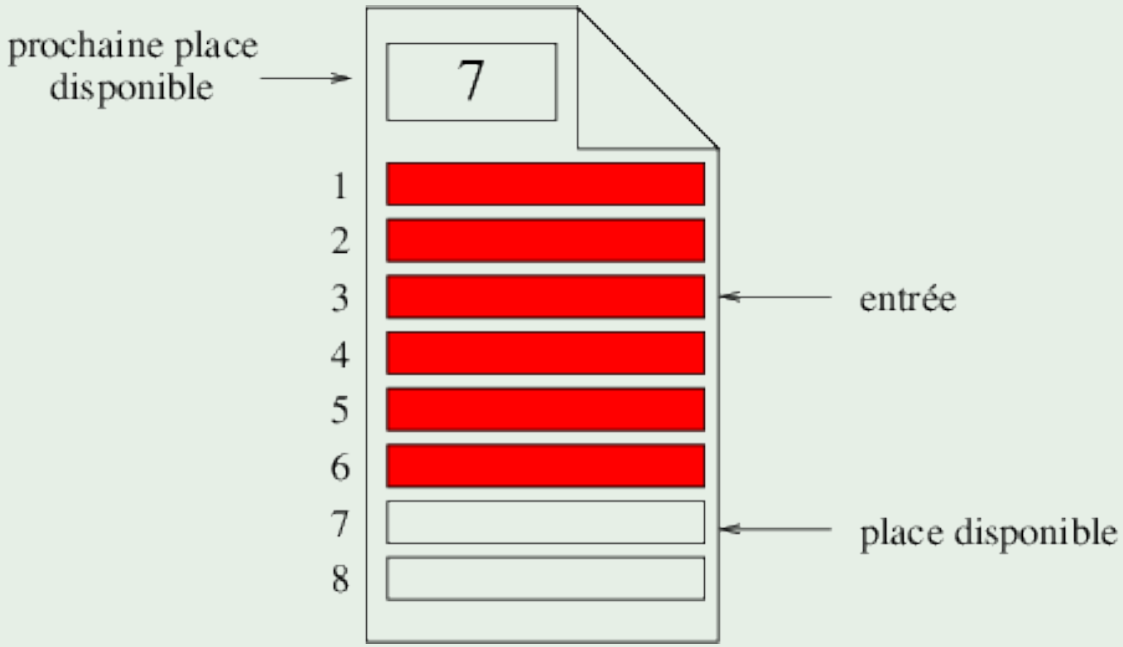
  ```
  mon_thread.join()
  ```

Table de matière

Fichier partagé (1/2)

Cas d'un annuaire partagé

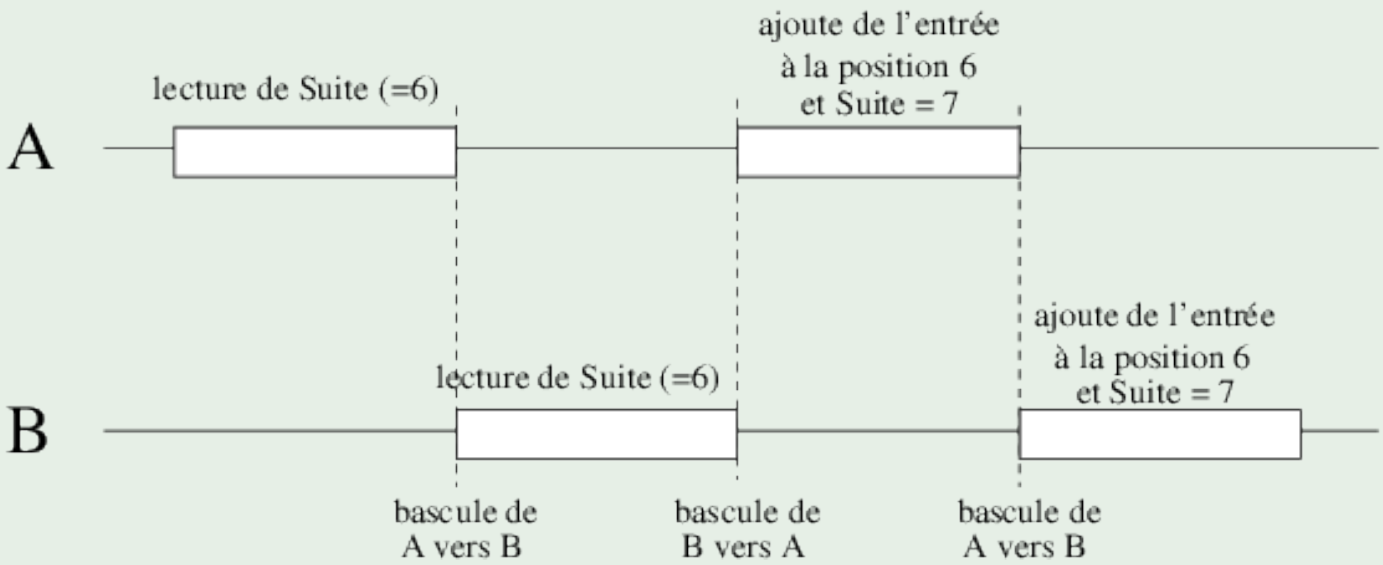
- Supposons un annuaire stocké dans un fichier contenant plusieurs entrées de taille fixe :
  $\Longrightarrow$ L'annuaire est accessible par plusieurs processus (légers).
- Chaque entrée est stockée à une position donnée dans le fichier ;
- En tête de fichier, le numéro de la prochaine place disponible est indiqué (nous l'appellerons Suite).



Fichier partagé (2/2)

Scénario possible avec l'annuaire partagé

$\Longrightarrow$ Que se passe-t-il si deux processus A et B désirent ajouter simultanément une nouvelle entrée à l'annuaire ?



Sections critiques

Ressources critiques

Les ressources logicielles ou matérielles qui posent des problèmes sont dites **critiques**.

Section critiques

Les portions de code qui manipulent ces ressources critiques sont appelées des **sections critiques**. Ces sections doivent être exécutées en **exclusion mutuelle**

Utilité

Les notions de **ressource critique** et **section critique** sont utiles :

- pour les threads d'un processus utilisateur ;
- pour les données du système d'exploitation partagées par les processus.

Le problème de l'exclusion mutuelle

Forme des programmes

`<initialisation>` exécuté une seule fois

⋮

`<prologue>`

section critique

`<épilogue>`

Contraintes

- Il existe **au plus** un processus en section critique (S.C.) ;
- Les processus ne sont pas bloqués sans raison (absence de **privation**) ;
- Les sections `<prologue>` et `<épilogue>` sont les mêmes pour tous les processus (**uniformité**) ;
- Le blocage du processus en S.C. ne doit pas entraîner de privation (**tolérance aux pannes**).

Section critique

Scénario avec les sections critiques

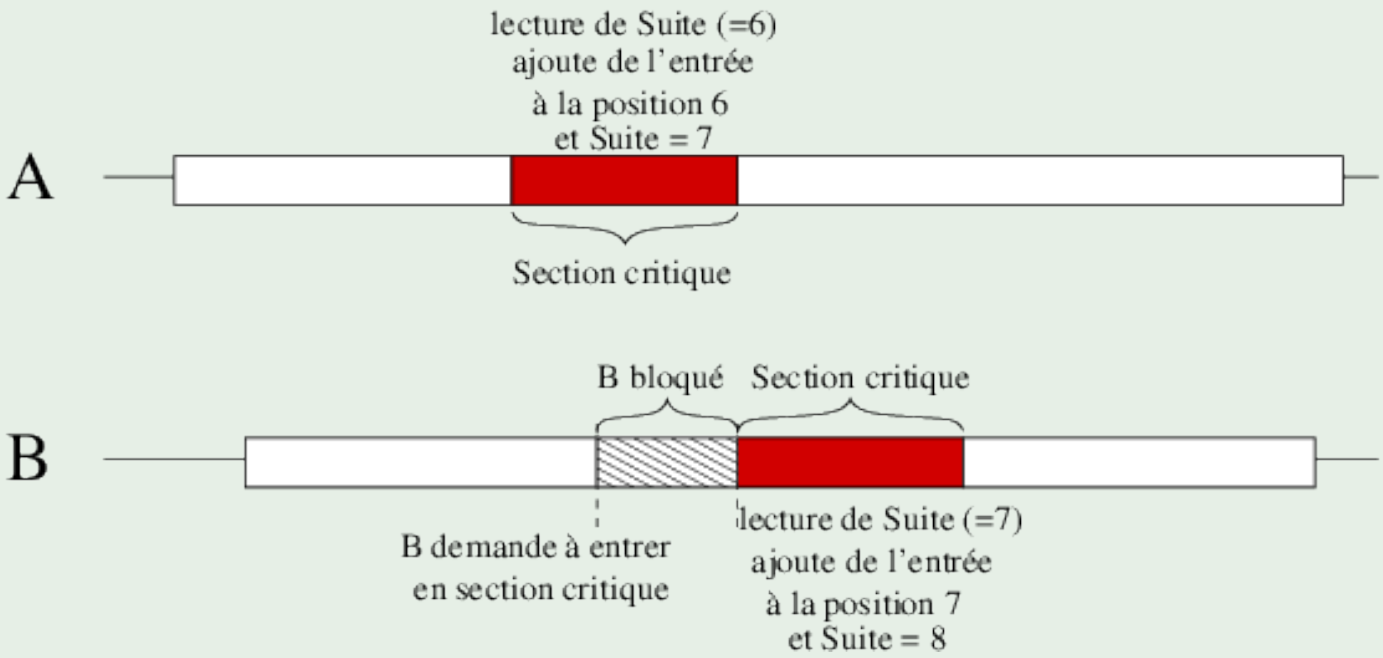


Table de matière

Solution de Peterson pour deux processus

Description

- Chaque processus est identifié par un numéro (ici 0 ou 1) ;
- Données partagées par tous les processus :
  - Variable tour ;
  - Tableau états : indique si le processus désire entrer en section critique.

Exemple de code

var tour : entier ;
états : tableau [0 .. 1] de booleen ;

Codage pour le processus P_i :

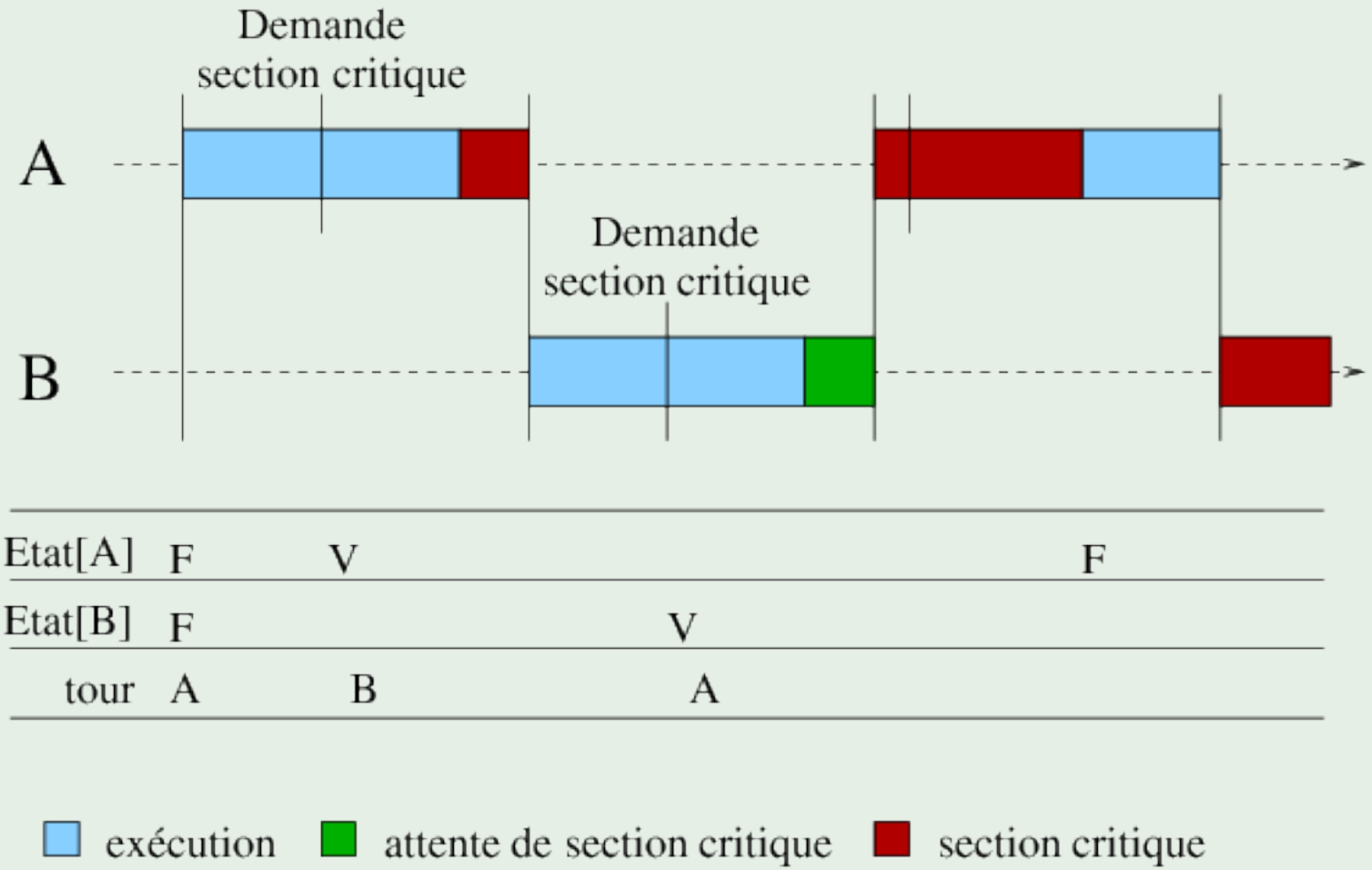
```
<initialisation>   (1)  tour := 0 ;
                   (2)  états := (faux, faux) ;

<prologue>         (3)  états[i] := vrai ;
                   (4)  tour := 1 − i ;
                   (5)  repeter
                   (6)  jusqu'a (tour = i) ou (états[1 − i] = faux)

<épilogue>         (7)  états[i] := faux ;
```

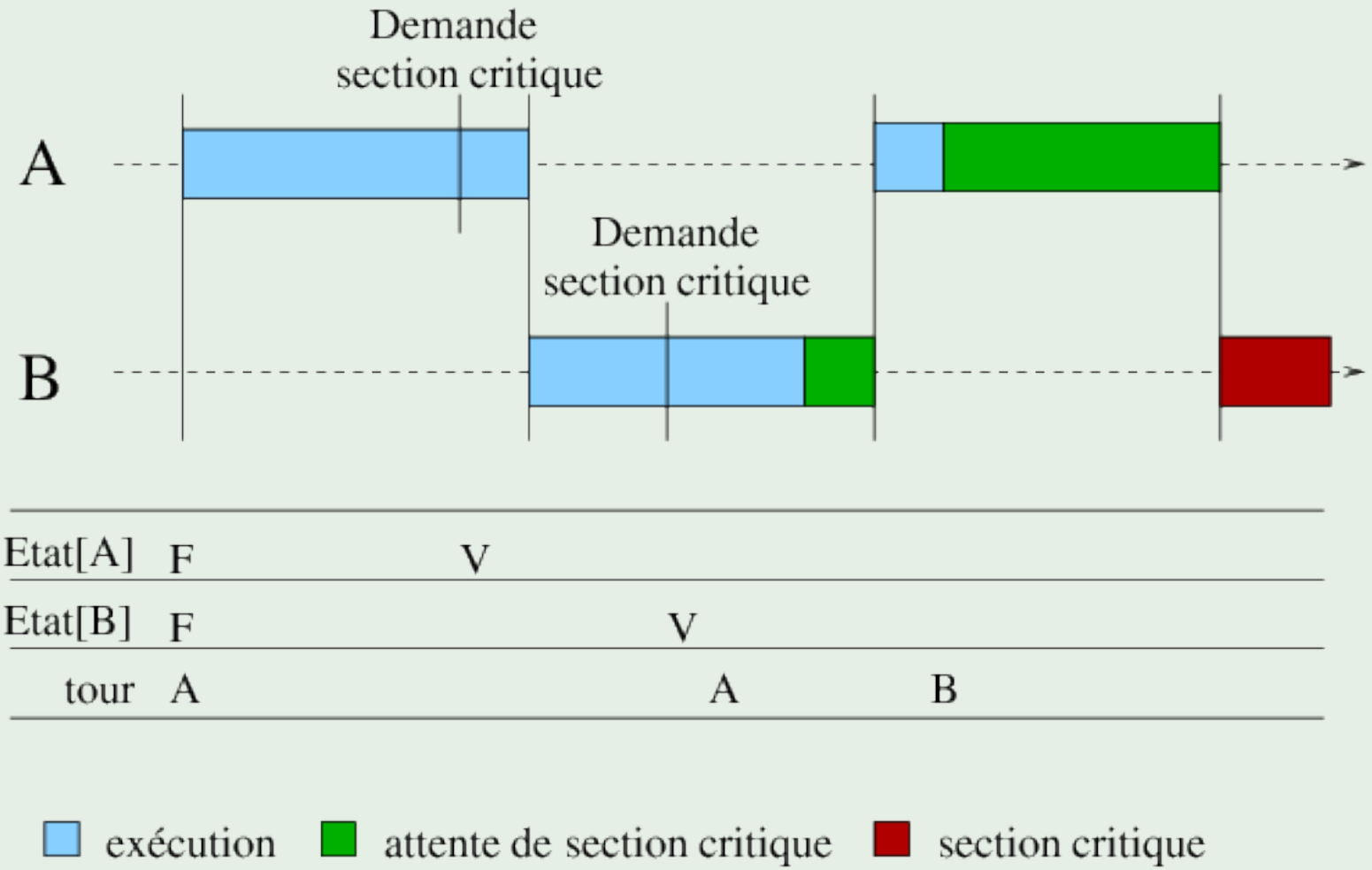
Exemple de l'algorithme de Peterson (1/2)

Exemple d'exécution



Exemple de l'algorithme de Peterson(2/2)

Exemple d'exécution

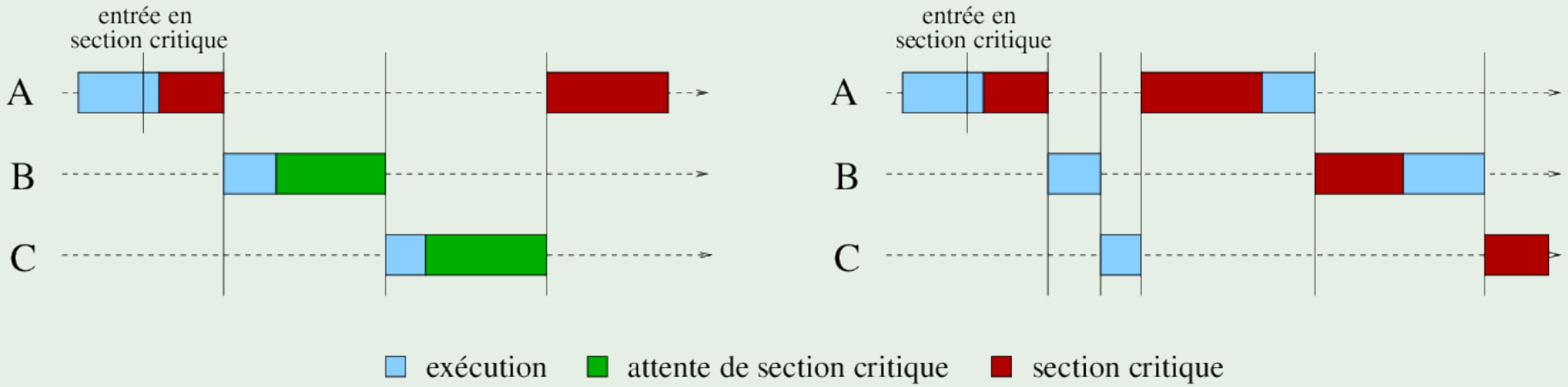


Attente active vs attente passive

Définitions

- *Attente active* : le processus est en attente de section critique mais consomme du CPU ;
- *Attente passive* : le processus est en attente de section critique mais ne consomme pas de CPU.

Exemples



Les verrous (1/4)

Objectifs

- ne plus perdre de temps CPU,
- simplicité de la solution.

Définition des verrous

Un **verrou** est une structure de donnée **partagée** du système d'exploitation.

verrou : { *libre* : booléen ; *f* : file FIFO de processus ; }

procedure init(var *v* : verrou)
  v.libre := vrai ;
  v.f := {} ;

Les verrous (2/4)

Exemple de code

Pour un verrou donné, les deux procédures ci-dessous s'exécutent en exclusion mutuelle.

```
procedure prendre(var v : verrou)
    si (v.libre = faux) alors
        soit P le processus appelant
        entrer P dans la file v.f
        suspendre le processus P
    sinon
        v.libre := faux ;
    finsi

procedure libérer(var v : verrou)
    si la file v.f est vide alors
        v.libre := vrai ;
    sinon
        v.libre := faux ;
        sortir un processus Q de la file v.f
        réveiller le processus Q
    finsi
```

Les verrous (3/4)

L'exclusion mutuelle avec les verrous

Soit

var mutex : verrou ;

Le code de l'exclusion mutuelle s'écrit

```
<initialisation>   (1)   init(mutex) ;

<prologue>         (2)   prendre(mutex) ;
                         ⋮
                         section critique
                         ⋮
<épilogue>         (3)   libérer(mutex) ;
```

Les verrous (4/4)

Difficultés des verrous

Soit deux processus qui partagent deux ressources :

```
var   mutex1 : verrou ;
      mutex2 : verrou ;

      init(mutex1) ;
      init(mutex2) ;
```

P_1	(1)	prendre(*mutex1*) ;	P_2	(1')	prendre(*mutex2*) ;
	(2)	prendre(*mutex2*) ;		(2')	prendre(*mutex1*) ;
	(3)	**section critique**		(3')	**section critique**
	(4)	libérer(*mutex2*) ;		(4')	libérer(*mutex1*) ;
	(5)	libérer(*mutex1*) ;		(5')	libérer(*mutex2*) ;

Il y a blocage pour la séquence **1 ... 1' ... 2 ... 2'**