CC BY NC SA

Processus légers et synchronisation

Vania Marangozova-Martin
Maître de Conférences, UGA
Vania.Marangozova-Martin@imag.fr

CC BY NC SA

Résumé de la séance précédente

- **Les processus = processus lourds**
  - Espace d'adressage dédié, gros contexte
- **Les threads : processus légers**
  - Partage de mémoire : communication et coordination faciles
  - Moins de ressources lors de la création : moins coûteux et plus rapide
  - Changement de contexte plus rapide
  - Parallélisme
  - Complexité de programmation
  - Pas de protection
  - Problèmes de synchronisation
  - L'accélération n'est pas linéaire

Problèmes de synchronisation (cf. plus loin)

- **Les threads partagent des données**
- **Les données (variables) peuvent donc être lues et modifiées**
- **Ces deux modes d'accès (LECTURE/READ et ÉCRITURE/WRITE) sont conflictuels**

- **L'exemple de ReaderThread + WriterThread**

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_threads.py
Starting the main thread!!!
Exiting the main thread with lx = [41.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 863.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -69.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 47.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0]
```

cc BY NC SA

Accélération

- **L'accélération (*speedup*)**
  - T_1 : temps séquentiel (sur 1 processeur)
  - T_p : temps parallèle (sur p processeurs)

$$S_p = \frac{T_1}{T_p}$$

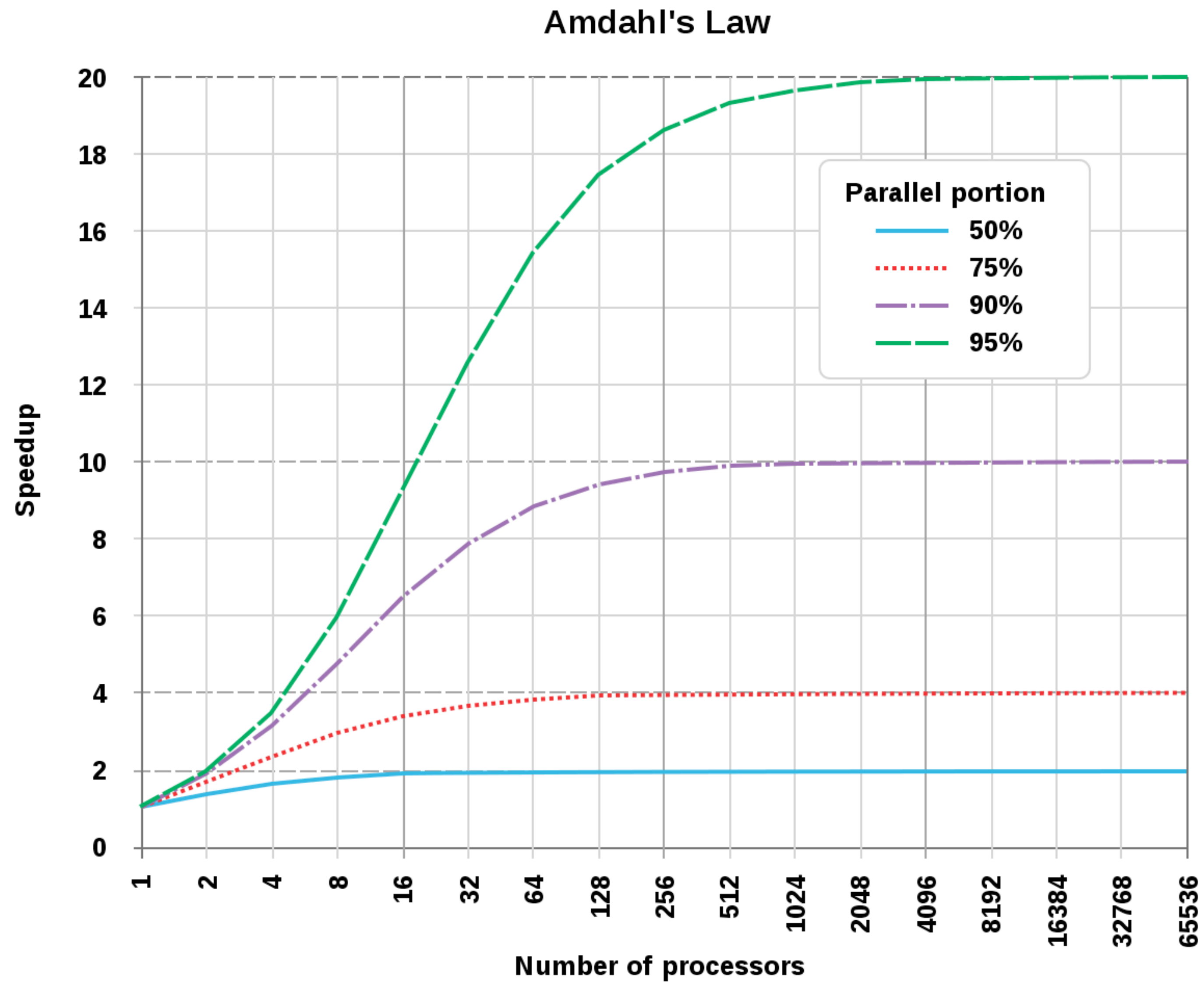
- **Loi d'Amdahl**
  - si z la proportion du temps d'exécution de la partie de programme pouvant être parallélisée
  - p le nombre d'unités de calcul (= accélération de la partie parallélisée)

$$S_p = \frac{1}{(1-z) + z/p}$$

  - si z = 50% et p = 2, Sp = 1, 333...
  - si z= 90% et p = 9, Sp = 5!

CC BY NC SA

https://fr.wikipedia.org/wiki/Loi_d%27Amdahl#/media/Fichier:AmdahlsLaw.svg



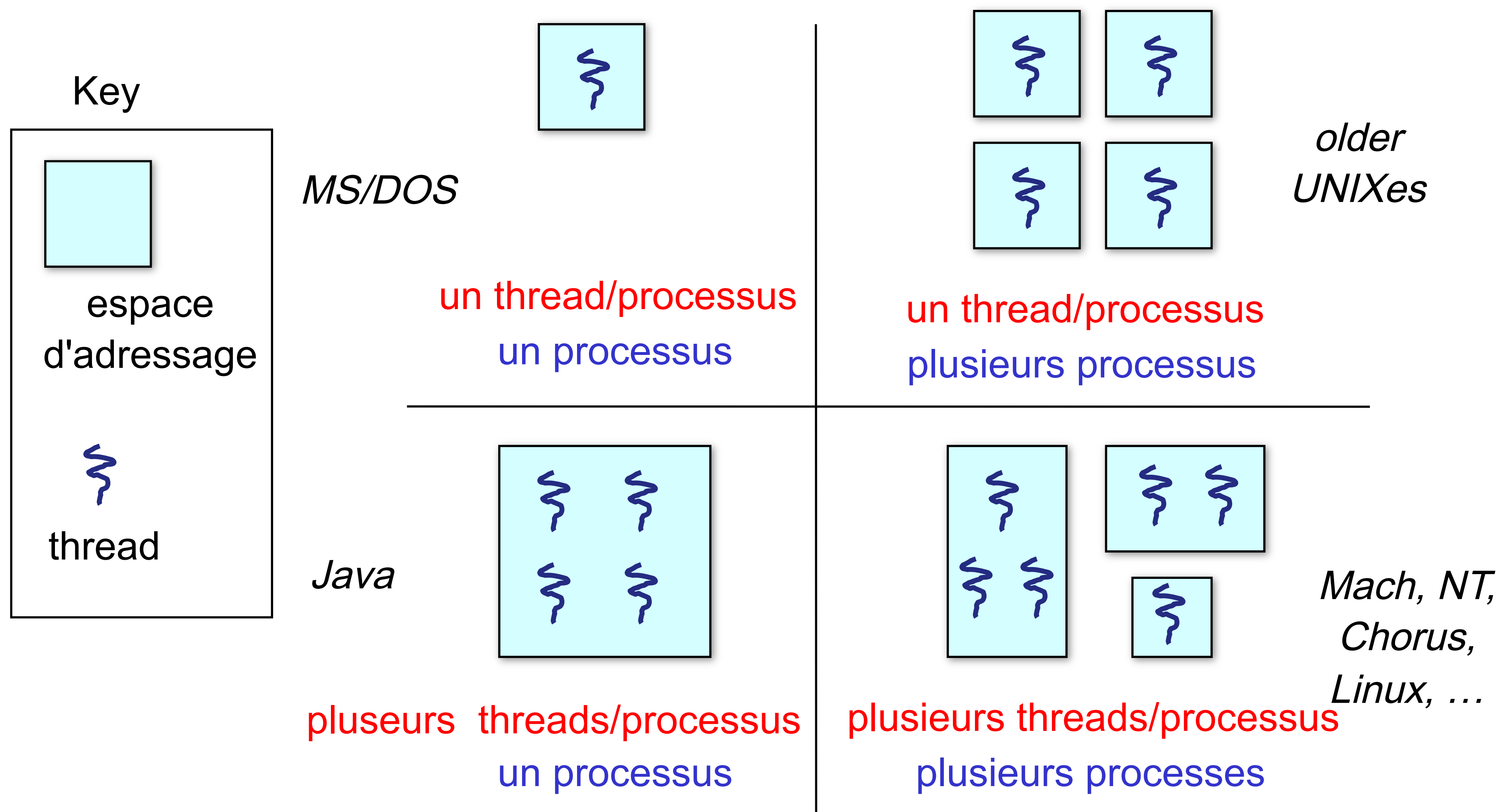
Implémentation des *threads*

CC BY NC SA

- **Les processus lourds sont gérés (implémentés) par le SE**
  - c'est le SE qui implémente les fonctions de gestion (création, terminaison, changement d'état, ordonnancement, ...)
  - c'est le SE qui alloue les ressources (mémoire, ...)
  - c'est le SE qui maintient le contexte d'un processus
- **Qui gère les processus légers?**
  - Il existe des implémentations où c'est le système
  - et d'autres où c'est une bibliothèque utilisateur

cc BY NC SA

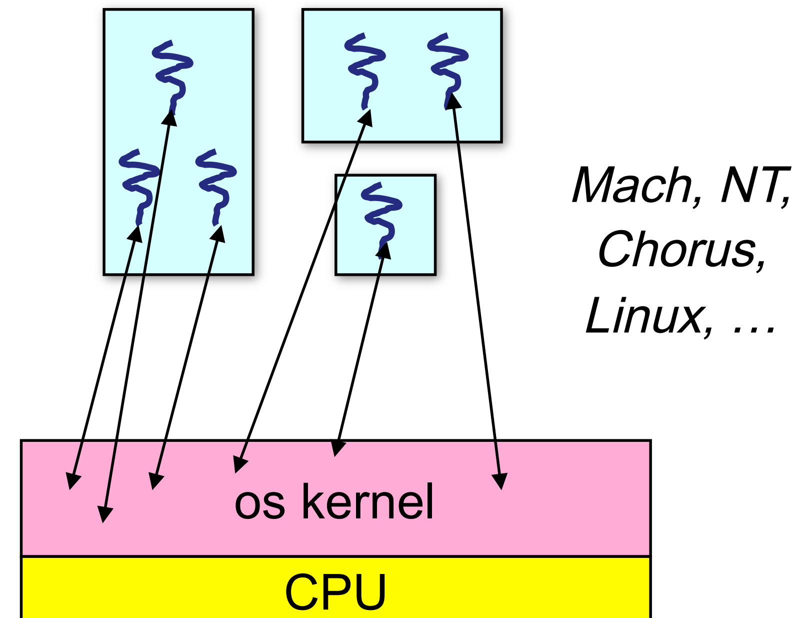
Implémentation des threads



cc BY NC SA

Implémentation au niveau système : threads noyau

- **Le SE gère des threads et des processus**
  - toutes les opérations liées aux threads sont implémentées dans le noyau



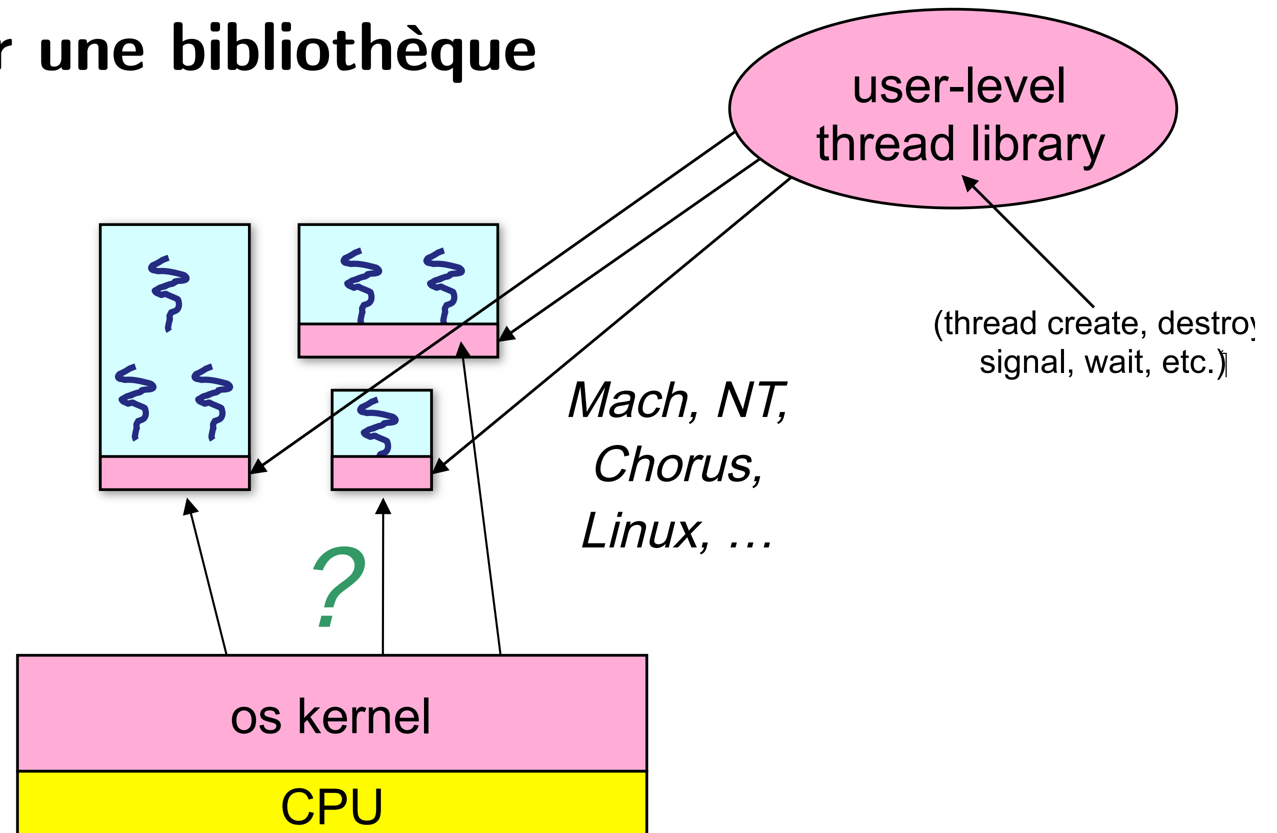
Mach, NT, Chorus, Linux, ...

  - si un thread est bloqué, le SE peut ordonnancer un autre thread
    - recouvrement entre les E/S et le calcul dans un processus
  - threads moins coûteux que les processus
  - coûteux quand même puisque appels système : x 100 cycles

(cc) BY NC SA

Implémentation au niveau utilisateur: threads user

- **Implémentation par une bibliothèque user qui gère tout**

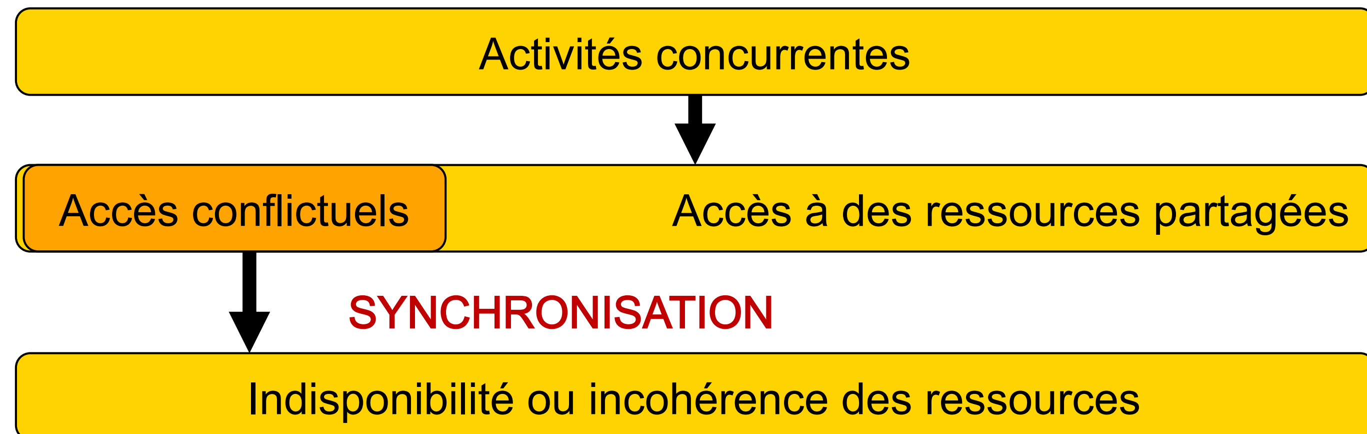


- toutes les opérations sont des appels de fonctions i.e. x 1 cycle!
- le SE ne connaît pas les threads et les threads bloquant sont problématiques
- la bibliothèque peut être portable ou non (~SE)

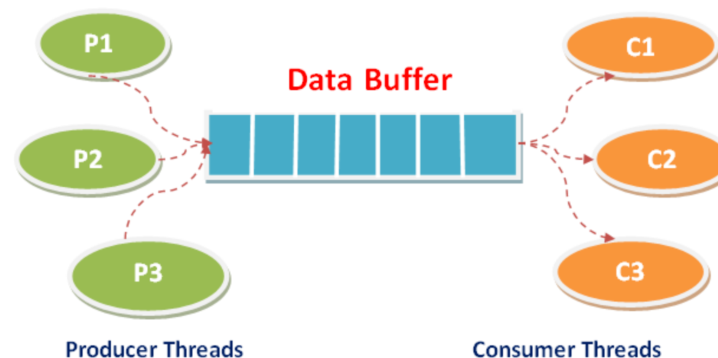
Synchronisation

CC BY NC SA

- **Quand ?**



- **Compétition et coordination**
  - Compétition
    - accès à des ressources partagées : fichiers, variables, imprimantes, processeur,...
  - Coordination
    - ex. producteur-consommateur



Propriétés de la solution de synchronisation

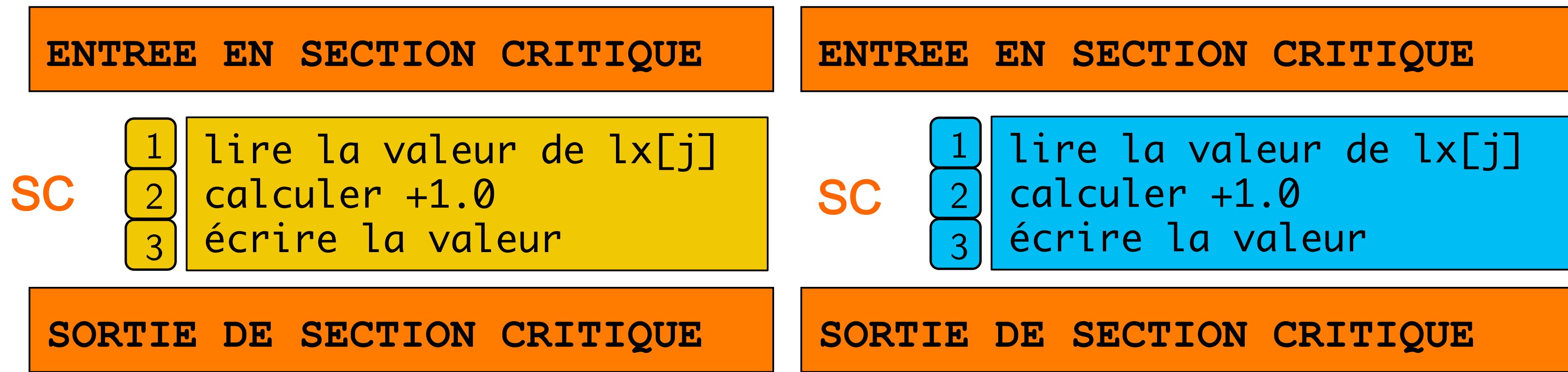
CC BY NC SA

- **Équité**
  - Tous les processus ont-ils les mêmes chances d'accéder à la ressources/ de faire?
- **Famine**
  - Tous les processus accèdent-ils à la ressource au bout d'un temps fini?
- **Interblocages**
  - Situation de non-avancement? Processus s'attendent-ils mutuellement?
- **Performances**
  - Combien de temps est-il "perdu" en synchronisation?

La section critique (Djikstra)

CC BY NC SA

- **Elément de synchronisation de base**

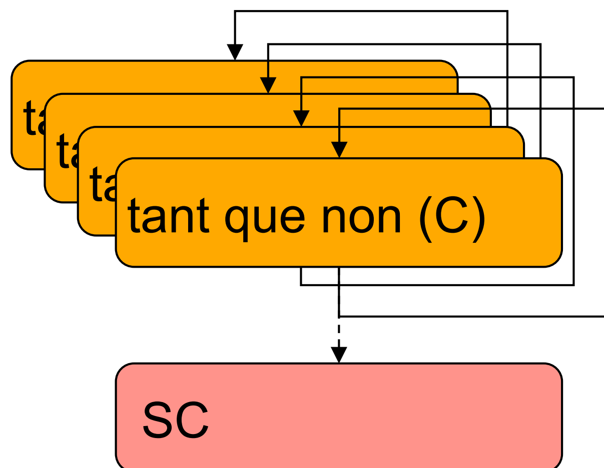


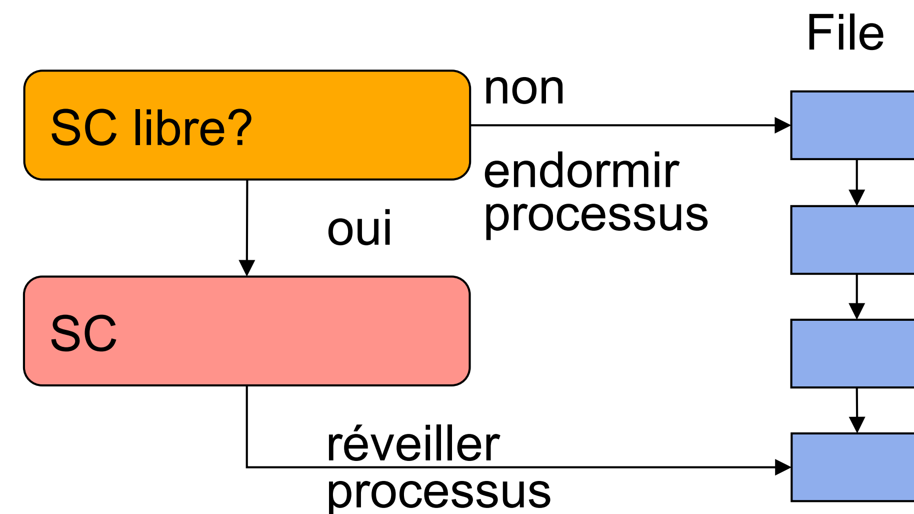
- **Code de section critique**
  - Indivisible, atomique
  - Exécuté par un seul processus (exclusion mutuelle)
  - Indépendamment de la vitesse du CPU
  - Pas de blocage, de privation

CC BY NC SA

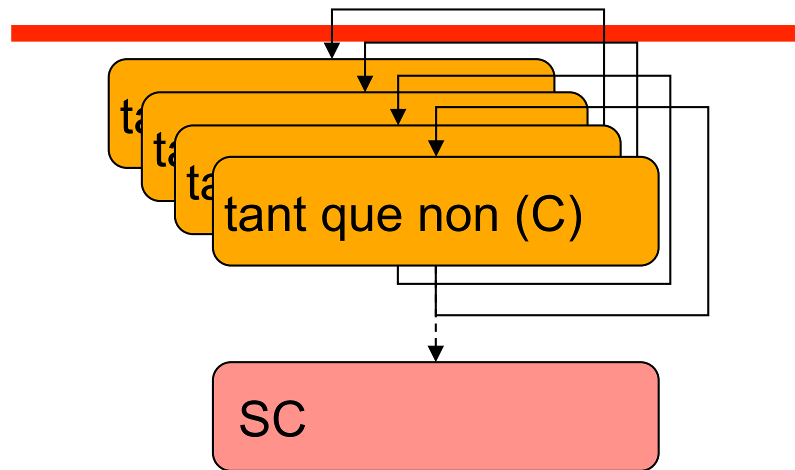
Mise en place des sections critiques

- **Attente active**
  - Boucle et test
  - Inefficace en CPU
- **Attente passive**
  - Processus bloqué si SC occupée
  - Processus débloqué si SC libérée





Attente active



Solutions logicielles avec attente active

▶ Algorithme de Lamport

```
int number[n+1] = {0,0,0,…}; //données partagées
bool choosing[n+1];
Pi (i=0..N):
  choosing[i] = true;
  number[i] = 1 + max(number[0], …, number[n-1]);
  choosing[i] = false;
  for (j=1; j<= n; j++) {
    while (choosing[j]); // attend que Pj ai un numéro
    while (number[j] != 0) && //Pj veut SC & Pj +prioritaire
          ((number[j] < number[i]) ||
          (number[j] == number[i]) && j<i)));}

  SECTION CRITIQUE
  number[i] = 0;
```

Solution matérielle avec attente active : Test & Set

- **Instruction Test & Set : atomique**

```
def TestAndSet(adresse):
    rv = lire la valeur à l'adresse
    positionner la valeur à l'adresse à 1
    return rv
```

CC BY NC SA

Solution matérielle avec attente active : Exemple en Python

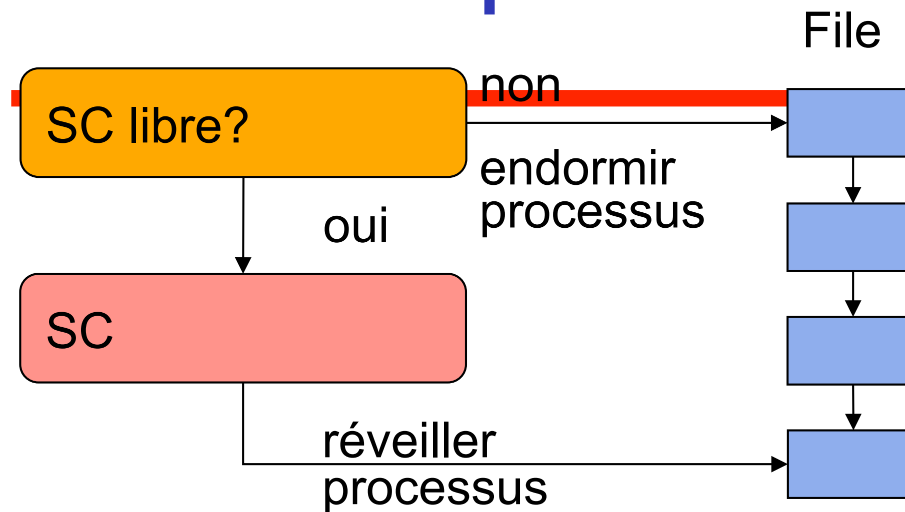
- **Instruction Test & Set : atomique**

```
def TestAndSet(verrou):  #verrou est passé par référence
    rv = verrou.val         #lire la valeur de son attribut
    verrou.val = 1          #modifier son attribut
    return rv               #retourner la valeur initiale
```

- **Verrou**

```
class Verrou():
    def __init__(self):
        self.val = 0
    def lock(self):
        while TestAndSet(self) == 1:
            pass
    def unlock(self):
      self.val = 0
```

Attente passive



CC BY NC SA

Verrou

- **Synchronisation avec attente incoditionnelle**
- **Trop simple!**
  - Les opérations
    - `verrouiller` (= prendre le verrou)
    - `déverrouiller` (= relâcher le verrou)

Exemple : parking N places

CC BY NC SA

```
int N = ...; //nombre of places

//code non synchronisé
entrer_parking ( )                sortir_parking( )
  while (N==0) ;//attendre          N++;
  N--;
```

```
//code synchronisé à l'aide de verrou, INCORRECT
verrou = Threading.Lock() // pour manipuler N

def entrer_parking ( ):           def sortir_parking( ):
  verrou.acquire()                   verrou.acquire()
  while (N==0)                       N++;
  N–                                 verrou.release()
  verrou.release()
```

monopolisation du verrou
!!interblocage!!

Exemple : parking N places

```
//code synchronisé à l'aide de mutex CORRECT
verrou = Threading.Lock()

entrer_parking ( )
  verrou.acquire()
  while (N==0):
    verrou.acquire()
    verrou.release()
  N--;
  verrou.release()

sortir_parking( )
    verrou.acquire()
    N++;
    verrou.release()
```

```
int N = ..;
entrer_parking ( )
  while (N==0) <attendre>
  N--;

sortir_parking( )
    N++;
    if (processus attend)
       <réveil processus>;
```

BY NC SA

Section critique conditionnelle (SCC)

cc BY NC SA

- **Verrous + files d'attente (*variables conditionnelles*)**
- **Primitives**
  - ATTENTION : les primitives, leurs noms et leurs syntaxe exacte dépendent du langage de programmation
  - **`lock(verrou)`**
  - **`wait(verrou, condition)`**
    - exécution en deux phases

```
1) libère le verrou
2) bloque le thread
   courant sur condition
```

```
1) réveil du thread
```

```
2) reprendre verrou
```

  - **`signal(verrou, condition)`**
    - réveille un thread bloqué sur condition, s'il y en a un.
    - si pas de thread en attente, le signal est perdu
  - **`broadcast(verrou, condition)`**
    - réveille tous les threads en attente sur la condition
  - **`unlock(verrou)`**

BY NC SA

Problème du parking : solution avec SCC

- **En pseudo-code**

```
SCC Parking {
  condition fileV;
  lock v;
  int NB_LIBRES;


void entrer_parking()
    lock(v);
    while (NB_LIBRES == 0)
        wait (v, fileV);
    NB_LIBRES --;
    unlock(v);

}
```

```
void init (int nbPlaces)
  NB_LIBRES = nbPlaces;
  fileV.init()
  v.init()

void sortir_parking ()
  lock(v);
  NB_LIBRES ++;
  signal(v, fileV);
  unlock(v);
```

Problème du parking : en Python

CC BY NC SA

```python
#!/usr/bin/env python3
import time
import threading
import random

ITERATIONS = 2
NB_THREADS = 10
NB_PLACES_PARKING=2

# Parking -------------------------------------
class Parking():

    def __init__(self, nb_places):
        self.cond = threading.Condition()
        self.NB_PLACES=nb_places

    def entrer_parking(self):
        self.cond.acquire()
        while self.NB_PLACES == 0:
            self.cond.wait()
        self.NB_PLACES-=1
        print("... (NB_PLACES = %d) ...\n" % self.NB_PLACES
        self.cond.release()

    def sortir_parking(self):
        self.cond.acquire()
        self.NB_PLACES+=1
        self.cond.notify()
        self.cond.release()
```

```python
# Voiture -------------------------------------
class Voiture (threading.Thread):
    def __init__(self, number, parking):
        threading.Thread.__init__(self)
        self.number = number
        self.parking = parking

    def run(self):
        print("I am car number %d \n" % self.number
        for i in range(0,ITERATIONS):
            # entrer dans le parking
            print("-> %d WAITING\n" % self.number,)
            self.parking.entrer_parking()
            #rester garee
            print("... %d ...\n" % self.number )
            time.sleep(random.randint(2,20))
            # sortir du parking
            print("<- %d \n" % self.number,)
            self.parking.sortir_parking()
```

```python
# main function ---------------------------------
def main():

    # Creer le parking
    P = Parking(NB_PLACES_PARKING)

    # Create new threads
    threads = []
    for t in range(0,NB_THREADS):
        thread = Voiture(t, P)
        threads.append(thread)

    # Start new threads
    for t in threads:
```

Problème du parking : retour sur la solution avec SCC

- **Le vol de cycles : le cas du "if"**

(1) **le thread T se bloque**

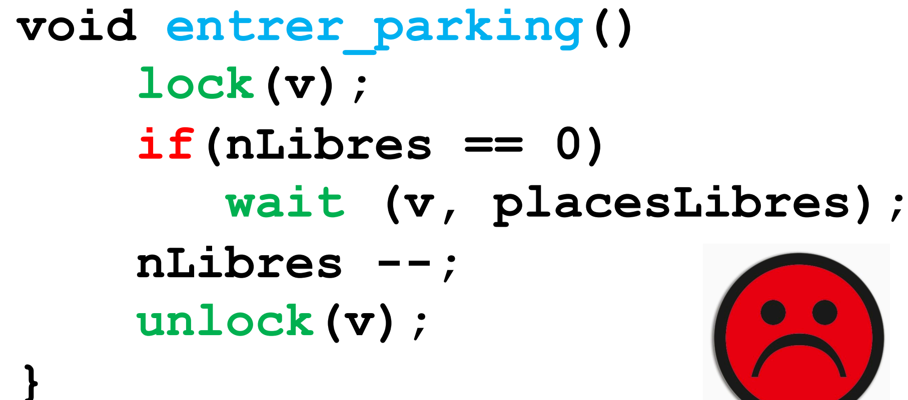
(2) **un thread S sort et libère une place**

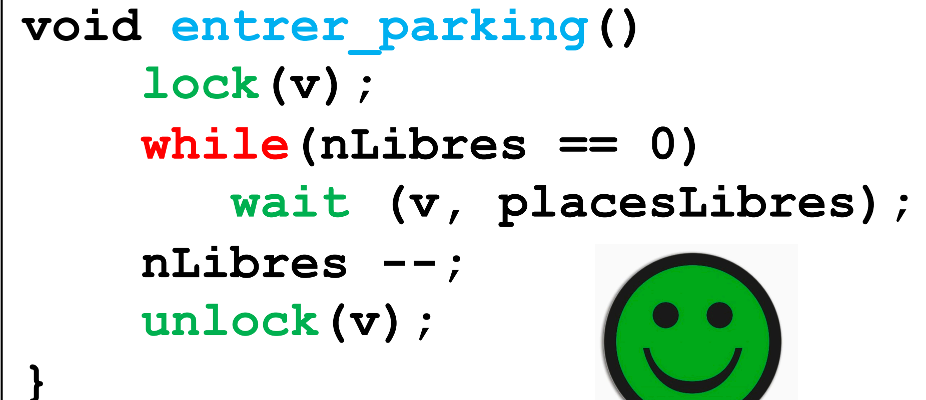
(3) **le thread T est réveillé**

(4) **il essaie d'acquérir le verrou mais est préempté**

(5) **un autre thread M prend le verrou, vérifie la condition et prend la place**

(6) **T acquière le verrou, ne vérifie pas la condition et rentre dans le parking :(**





Deux files d'attente

```
SCC Parking {
  condition fileNormale;
  condition fileVIP;
  lock v;
  int nLibres, VIPattentes;


void entreeNormale_parking()
    lock(v);
    while (VIPattente > 0 or
         nLibres == 0)
        wait (v, fileNormale)
    nLibres -=1
    unlock(v);
}
```

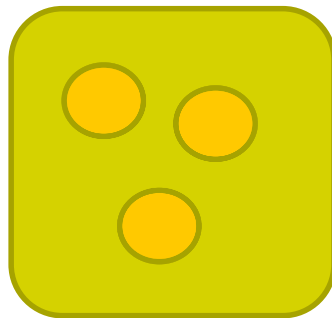
```
void sortie_parking()
    lock(v);
    if (VIPattente > 0)
        signal(v, fileVIP);
    else
        signal(v, fileNormale);
    nLibres ++;
    unlock(v);
}
void entreeVIP_parking()
    lock(v);
    VIPattente+=1
    while (nLibres == 0)
        wait (v, fileVIP);
    nLibres --;
    VIPattente-=1
    unlock(v);
}
```

cc BY NC SA

Sémaphore (Dijkstra, 1965)

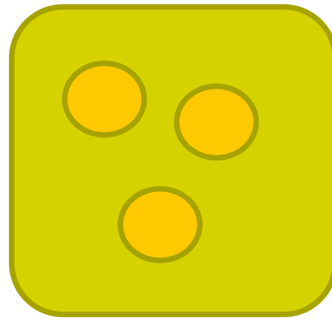
- **Compteur : nombre de processus en SC**
- **File d'attente**
- **Opérations**
  - P : entrée en SC (wait)
  - V : sortie en SC (signal)

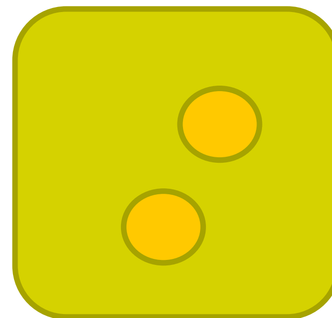
- **Pour comprendre le fonctionnement : analogie avec une boîte à jetons**
  - chaque thread doit obtenir un jeton pour pouvoir aller en SC



Sémaphore = boîte à jetons

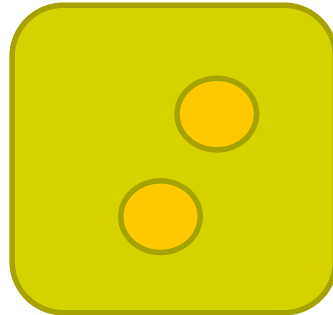
CC BY NC SA

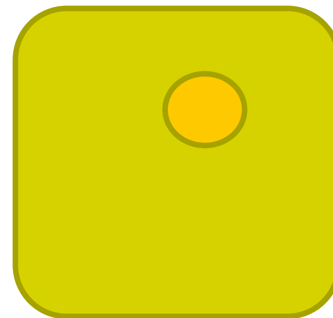




cc BY NC SA

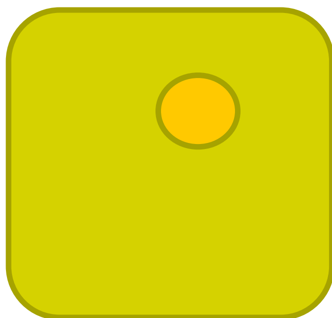
Sémaphore = boîte à jetons

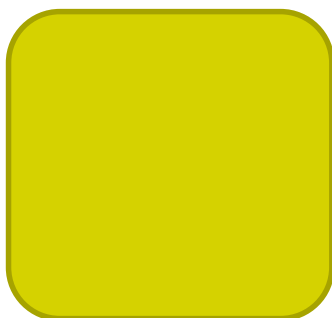




Sémaphore = boîte à jetons

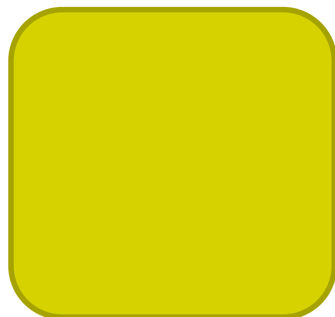
CC BY NC SA

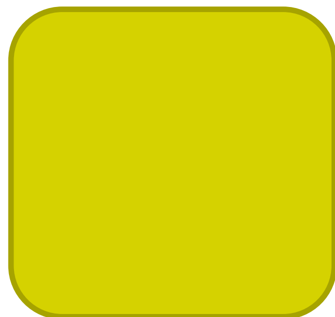




Sémaphore = boîte à jetons

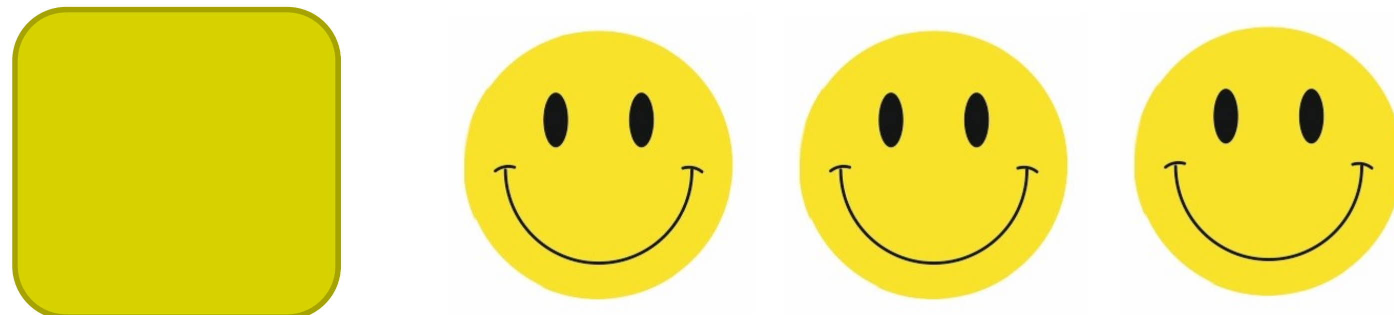
cc BY NC SA





cc BY NC SA

Sémaphore = boîte à jetons



CC BY NC SA

Parking avec sémaphore

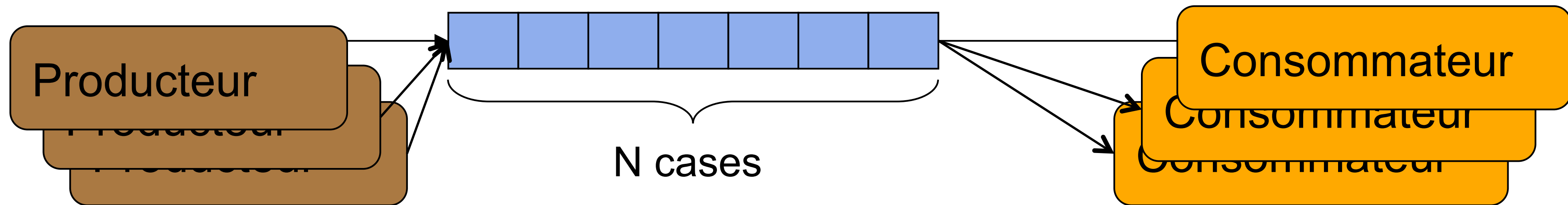
```
int N = ...;
Semaphore S = new Semaphore(N);

entrer_parking ( )        sortir_parking( )
  P(S);                       V(S)
```

- **En Python ce n'est pas bien plus compliqué**

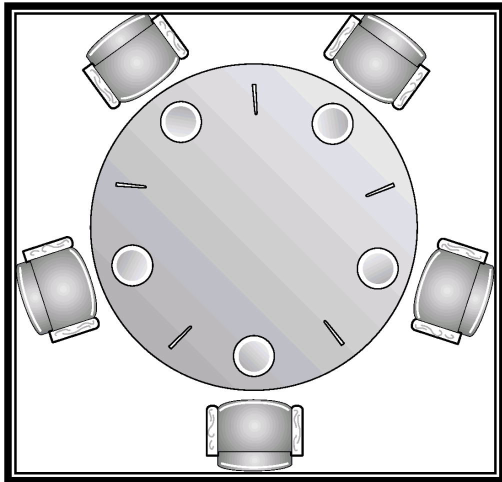
```python
# Parking -------------------------------------
class Parking():

    def __init__(self, nb_places):
        self.sem = threading.Semaphore(nb_places)

    def entrer_parking(self):
        self.sem.acquire()

    def sortir_parking(self):
        self.sem.release()
```

Problèmes de synchronisation classiques

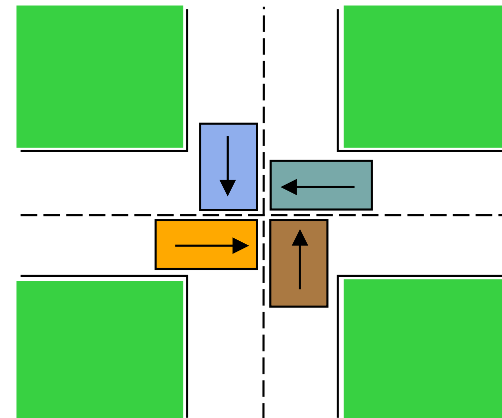
- **Producteur-consommateur**



- **Comptes bancaires : débit/crédit**
- **Les philosophes**



- **Les feux/les intersections**



A vous

cc BY NC SA

- **Producteur – consommateur**
  - Deux types de threads
  - Tous les threads travaillent avec le même tampon
  - Les producteurs produisent i.e. remplissent les cases du tampon
    - Les producteurs sont bloqués quand ?
  - Les consommateurs vident les cases du tampon
    - Les consommateurs sont bloqués quand?
- **A vous d'écrire les fonctions qui modélisent le fonctionnement des threads**

```
# PROD -------------------------------------
class Producteur (threading.Thread):
    def __init__(self, number, tampon):
        threading.Thread.__init__(self)
        self.number = number
        self.tampon = tampon

    def run(self):
        print("I am PROD number %d \n" % self.number)
        for i in range(0,ITERATIONS):
            # remplir une case
```

```
# CONS -------------------------------------
class Consommateur (threading.Thread):
    def __init__(self, number, tampon):
        threading.Thread.__init__(self)
        self.number = number
        self.tampon = tampon

    def run(self):
        print("I am CONS number %d \n" % self.number)
        for i in range(0,ITERATIONS):
            # vider une case
```