Programmation concurrente et distribuée

Sylvain Conchon

Laboratoire Méthodes Formelles

Université Paris-Saclay, CNRS, ENS Paris-Saclay

sylvain.conchon@universite-paris-saclay.fr

Contenu du cours

- Introduction
- Programmation avec des Threads
- Programmation distribuée avec des sockets
- Quelques problèmes d'algorithmique distribuée : le problème du consensus, la notion d'horloge globale

Évaluation par mini projets

INTRODUCTION

Systèmes Concurrents (1)

Il s'agit de systèmes informatiques où plusieurs tâches s'exécutent au même moment.

Certaines tâches peuvent coopérer étroitement pour résoudre un problème, tandis que d'autres sont plus indépendantes mais partagent (et se disputent) les ressources du système.

L'architecture de ces systèmes peut prendre plusieurs formes :

- machine uniprocesseur ;
- machine multi-processeurs avec mémoire partagée ;
- réseaux d'ordinateurs *avec* mémoire distribuée ;
- réseaux d'ordinateurs *sans* mémoire distribuée.

Systèmes Concurrents (2)

Les systèmes qui doivent réagir à des actions simultanées de leur environnement sont **naturellement** concurrents.

- systèmes *temps réel* (ex. contrôle de procédés) ;
- systèmes *transactionnels* (ex. systèmes de réservations) ;
- systèmes d'*exploitation*

D'autres sont **potentiellement** concurrents. Ce sont ceux qui peuvent bénéficier d'une implantation concurrente pour réduire les *délais d'attente*, ou pour aller *plus vite*, ou parce que le problème posé est plus facile à résourdre avec un *algorithme concurrent*.

Systèmes Distribués (Répartis)

Il s'agit de systèmes concurrents dont les tâches, réparties sur un réseaux d'ordinateurs, communiquent et se synchronisent par **échange de messages** (pas de mémoire partagée).

Les principales *caractéristiques* de ces systèmes sont :

- Absence d'horloge globale ;
- Absence d'état global ;
- Pannes partielles.

Certains systèmes sont *naturellement* distribués parce que leurs données, calculs, ou utilisateurs sont distribués.

Exemples : Web, Blockchain, Cloud, Jeux,

Programmation concurrente et distribuée

La programmation des systèmes concurrents et distribués nécessite des mécanismes pour :

- **Gérer** des tâches concurrentes (créer, exécuter, arrêter, fixer des priorités d'exécution) ;
- **Communiquer** (échange de messages, mémoire partagée) ;
- **Synchroniser** les activités des tâches concurrentes ;
- **Partager** les ressources du système ;
- **Protéger** l'accès à ces ressources.

Nous illustrerons ces concepts à travers le langage Python

PROGRAMMATION CONCURRENTE AVEC DES THREADS

Processus

La notion de tâche, appelée *traditionnellement* **processus**, représente un programme séquentiel qui s'*exécute*. Chaque processus est représenté par :

- Un espace d'adressage (texte du programme,pile,tas) ;
- Des descripteurs de ressources, des *handlers* ;
- Des registres, signaux, attributs d'ordonnancement ;
- ...

Cette architecture ne permet pas facilement le partage des ressources entre processus et le coût du changement de contexte est important (sauvegarde et restauration de nombreuses informations).

Processus Légers (threads)

Une solution pour résoudre ces problèmes consiste à associer plusieurs *activités* à un même processus.

Ainsi, un processus est maintenant constitué d'un **environnement d'exécution** et d'une collection de **Threads** (processus légers).

Environnement d'exécution	Thread
Espace d'adressage (texte, tas)	Pile et registres (copie)
Descripteurs de ressources	Attributs d'ordonnancement
Handlers de signaux	Signaux (masqués et pendants)
Données de synchronisation	

Utilisation des threads

Il y a deux grandes familles d'utilisation des threads :

Attentes asynchrones : un programme doit continuer à fonctionner en attendant des informations provenant d'une interface graphique, de communications réseaux, d'un système de sauvegarde, etc.

Calculs intensifs : certaines opérations d'un programme sont indépendantes et peuvent donc être effectuées en parallèle

Programmer avec des threads

La programmation avec des threads est plus complexe que la programmation séquentielle.

Cela tient essentiellement aux problèmes liés à l'accès aux ressources partagées comme les synchronisations ou interblocages qui apparaissent parfois de manière aléatoire.

À cela s'ajoute le fait qu'il est très difficile de tester un programme avec des threads :

- difficultés à influencer l'ordre d'exécution des threads
- complexité à déclencher (de manière automatique) des évènements asynchrones provenant d'un réseau, d'une interface graphique

Modèle M :1 (Many-to-One)

Une façon d'implanter les *threads* consiste à les gérer dans l'espace utilisateur à l'aide d'une librairie spécialisées ou directement dans le langage de programmation.

Ces *threads* **Utilisateurs** sont alors inconnues du système d'exploitation.

Avantages	Inconvénients
Changement de contexte rapide	Un thread bloqué bloque tous les autres
Portable sur tous systèmes (librairie)	Aucun parallélisme possible
Ordonnancement programmable	

Modèle 1 :1 (One-to-One)

Le modèle 1 :1 résout les problèmes du modèle M :1 en affectant **un** *thread* noyau à **chaque** *thread* utilisateur.

Les threads noyau sont gérés directement par le noyau du système d'exploitation, en particulier c'est le système qui réalise leur ordonnancement.

Avantages	Inconvénients
Le blocage d'un thread ne bloque pas systématiquement les autres	Commutation plus lente
Parallélisme possible (SMP)	Ordonnancement Système

Modèle M :N (Many-to-maNy)

Le modèle M :N combine les deux modèles précédents. Différents *threads* utilisateurs sont multiplexés sur un nombre (inférieur ou égal) de *threads* noyau. Cette attribution est faite au niveau de la librairie de thread.

L'ordonnancement est à deux niveaux :

- La librairie détermine quel *thread* (au sein d'un groupe) doit s'exécuter dans un *thread* noyau ;
- Le système gère l'ordonnancement des *threads* noyau.

Les *threads* sont créés et gérés par la librairie (peu coûteux) et l'exploitation du parallélisme (SMP) est possible.

Les Styles de Threads

Les styles de Threads les plus utilisés sont :

- les *threads* POSIX (*Pthreads*) ;
- les *threads* Windows (Win32 *threads*) ;
- les *threads* propres à certains langage (comme Java).

Les *Pthreads* et les *Win32 threads* sont des librairies. Ils sont utilisés à travers une API propre au système d'exploitation.

Quand il s'agit de threads propres à un langage, alors ils sont directement intégrés comme des constructions du langage.

L'implantation de ces *threads* varie selon les plate-formes.

Threads Posix

Appelés également pthreads, il s'agit d'une API normalisée IEEE (1003-c)

POSIX : Portable Operating System Interface

API disponible sous Linux et Mac OS.

Threads en Python

Pour écrire nos programmes avec des threads en Python, on utilisera principalement la bibliothèque threading :

- Cette bibliothèque est basée sur pthreads sous Linux et Mac Os, et Win32 sous Windows.
- Bien qu'étant basé sur de telles bibliothèques, Python ne permet l'exécution que d'un seul thread à la fois par son interpréteur (Global Interpreter Lock)

Création de threads : la classe `threading.Thread`

Le module `threading` fournit une classe `Thread` pour manipuler des threads.

Ainsi, il y a deux manières de créer des threads en Python :

- En déclarant une nouvelle classe qui hérite de `threading.Thread`
- En instanciant directement cette classe

Création de threads par héritage

```
import threading

class MyThread(threading.Thread):

    def __init__(self,n,c):
        threading.Thread.__init__(self)
        self.cpt = n
        self.char = c

    def run(self):
        for i in range(self.cpt):
            print(self.char)

t1 = MyThread(100,'.')
t2 = MyThread(100,'#')

t1.start()
t2.start()
```

Création de threads par instanciation

```
import threading

def run(n,c):
    for i in range(n):
        print(c)

t1 = threading.Thread(target=run,args=(100,'#'))
t2 = threading.Thread(target=run,args=(100,'.'))

t1.start()
t2.start()
```

Création et fin des threads

Les threads ainsi créés s'exécutent en parallèle du programme principal lorsqu'ils sont lancés avec la méthode `start`

Un thread termine quand la fonction qu'il exécute (ou la méthode `run` quand il s'agit d'un héritage) se termine.

Un programme Python se termine quand tous ses threads sont terminés.

Threads démons

Un thread peut être déclaré comme démon lors de sa création.

```
import threading
import time

def run(n,c):
    time.sleep(1)
    for i in range(n): print(c)

t1 = threading.Thread(target=run,args=(100,'#'),\
                      daemon=True)
t1.start()
```

Les threads démons sont arrêtés (brutalement) quand le programme principal se termine.

En d'autres termes, un programme Python se termine quand il n'y a plus que des threads démons qui tournent.

Suspendre

Pour suspendre un thread en attendant la terminaison d'un autre thread, on utilise la méthode `join`.

```
import threading

def run():
    ...

t = threading.Thread(target=run, daemon=True)
t.start()
t.join()
```

Un appel `t.join()` rend la main immédiatement si le thread (représenté par) `t` est déjà terminé.

Mémoire partagée

Un des intérêts de la programmation avec threads plutôt qu'avec des processus est de pouvoir partager l'espace mémoire entre tous les threads. Par exemple, dans l'exemple suivant, les threads partagent la variable `x`.

```
import threading

x = 0
def run():
    global x
    x += 1

t1 = threading.Thread(target=run)
t2 = threading.Thread(target=run)

t1.start(); t2.start()
t1.join(); t2.join()

print(x)
```

Arrêter un thread (1/3)

En Python, il n'y a pas de façon "simple" pour arrêter un thread.

Certains langages proposent une méthode `kill` mais son usage n'est pas recommandé. En efet, le thread qui est serait tué ainsi le serait brutalement (comme les threads démons), dans n'importe quel état, par exemple en détenant des verrous (voir plus loin).

Une technique pour arrêter un thread est d'utiliser une variable partagée pour passer un message (d'arrêt ou non). Le thread qui s'exécute doit vérifier l'état de cette variable pour s'auto-terminer.

Arrêter un thread (2/3)

```
import threading

stop = False

def run():
    global stop
    while not stop: print('.')

t1 = threading.Thread(target=run)
t1.start()

input('')
stop = True
```

Arrêter un thread (3/3)

```
import threading

class MyThread(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)
        self._stop = False

    def stop(self):
        self._stop = True

    def run(self):
        while not self._stop: print('.')


t1 = MyThread()
t1.start()

input('')
t1.stop()
```

Environnement local (1/2)

Chaque thread possède un environnement local qui lui est propre (non partagé).

Cet environnement est accessible à l'aide de la fonction `threading.local()`.

Il s'agit d'un dictionnaire auquel on peut ajouter des variables.

Environnement local (2/2)

```
import threading
import random

def stop_thread(env):
    if random.randrange(100) == 5:
        env.stop = True

def run():
    global env
    env = threading.local()
    env.stop = False
    stop_thread(env)
    while not env.stop:
        print('.')
        stop_thread(env)

t = threading.Thread(target=run)
t.start()
```

Situation de compétition (1/2)

Le programme suivant n'affiche pas toujours le même résultat. Pourquoi ?

```
import threading

cpt = 0

def run(n):
    global cpt
    for i in range(n):
        cpt = cpt + 1

t1 = threading.Thread(target=run,args=(100000,))
t2 = threading.Thread(target=run,args=(100000,))

t1.start(); t2.start()
t1.join(); t2.join()
print('Compteur',cpt)
```

Situation de compétition (2/2)

Dans l'exemple précédent, la valeur finale de `cpt` (quand les deux threads ont terminé) n'est pas toujours 20000.

Situation de compétition (2/2)

Dans l'exemple précédent, la valeur finale de `cpt` (quand les deux threads ont terminé) n'est pas toujours 20000.

Cela est dû au fait que les deux threads sont en compétition (en anglais, race condition) pour l'accès, en lecture et écriture, à la variable partagée `cpt`.

Selon l'ordonnancement des threads, l'affectation `cpt = cpt + 1`, peut être interrompue entre le moment où elle lit le contenu de la variable `cpt` et celui où elle affecte cette variable `cpt = ...`.

Sections Critiques

Le problème de l'exemple précédent provient du fait que l'opération de mise à jour `cpt = cpt + 1` n'est pas **atomique**

- Le partage des données entre *threads* pose donc un problème de maintien de cohérence
- Des *threads* en concurrence ont parfois besoin d'accéder de manière exclusive à des ressources critiques : c'est le problème de l'**exclusion mutuelle**
- Les portions de code nécessitant cet accès exclusif sont appelées **sections critiques**.

Synchronisation

De nombreuses solutions ont été proposées pour résoudre le problème de l'exclusion mutuelle (algorithmes, test&set, échange de valeurs, verrous&variables conditionnelles, sémaphores, moniteurs etc.).

Dans ce cours, nous allons présenter le système des verrous, des conditions et des sémaphores.

Verrous

Pour protéger l'accès à une donnée partagée, on peut créer des verrous en instanciant la classe `threading.Lock`.

```
import threading

v = threading.Lock()
```

Ces objets disposent des deux méthodes `acquire`, pour prendre le verrou, et `release`, pour le relacher.

Une propriété fondamentale d'un verrou est qu'il ne peut être possédé que par un thread à la fois

Section critique

Pour protéger une section critique (constituée d'un ensemble d'instructions), il suffit de créer un verrou `v` et d'appliquer le motif de programmation suivant :

```
v.acquire()

# section critique

v.release()
```

Comme un verrou ne peut être possédé que par un thread à la fois, `v.acquire()` bloque le thread qui fait cet appel tant que le verrou n'est pas disponible. Quand le verrou est libre, cet appel bloque le verrou.

Un verrou `v` ne peut être débloqué que par un appel `v.release()`. En particulier, la fin d'un thread ne libère pas les verrous qu'il a pris.

Section critique : exemple

```
import threading

v = threading.Lock()
cpt = 0

def run(n):
    global cpt
    for i in range(n):
        v.acquire()
        cpt = cpt + 1
        v.release()

t1 = threading.Thread(target=run,args=(100000,))
t2 = threading.Thread(target=run,args=(100000,))

t1.start(); t2.start()
t1.join(); t2.join()

print('Compteur',cpt)
```

Construction `with`

Le langage Python fournit une construction pour la prise de verrou avec une libération implicite

```
with lock:
    inst_1
    inst_2
    ...
    inst_k
```

Dans cet exemple, le verrou `lock` est pris pour exécuter un bloc d'instructions constitué de `inst_1`, ..., `inst_k`. Le verrou est libéré dès que le programme sort de ce bloc.

Ré-entrance (1/2)

Les verrours créés avec la classe `Lock` ne sont ré-entrants.

Par exemple, le programme suivant se bloque.

```
import threading

v = threading.Lock()
cpt = 0

def toto():
    global cpt
    v.acquire()
    cpt = cpt + 1
    v.release()

def run(n):
    with v:
        for i in range(n): toto()

t = threading.Thread(target=run,args=(100000,))
t.start(); t.join()
```

Ré-entrance (1/2)

Avec la classe `RLock`, Python fournit des verrous ré-entrants pour que la prise d'un verrou ne soit pas bloquante lorsqu'il s'agit du même thread d'exécution

```
import threading

v = threading.RLock()

def f():
    with v:
        g(); h()

def g():
    with v:
        h(); print('do g')

def h():
    with v:
        print('do h')

f()
```

Interblocages

La manipulation des verrous est délicate. Des interblocages sont possibles, comme dans l'exemple ci-dessous.

```
import threading

a = threading.Lock()
b = threading.Lock()

def f1():
    a.acquire()
    b.acquire()
    b.release()
    a.release()

def f2():
    b.acquire()
    a.acquire()
    a.release()
    b.release()

t1 = threading.Thread(target=f1)
t2 = threading.Thread(target=f2)
```

Conditions

Pour gérer finement le partage des ressources, il arrive parfois qu'une certaine condition soit satisfaite pour continuer l'exécution d'un thread.

Conditions : exemple (1/2)

Prenons l'exemple d'un buffer `b` avec une seule case. Ce buffer dispose de deux méthodes `put` et `get`.

Un appel `b.put(v)` permet de déposer une valeur `v` dans le buffer, à la condition que le buffer soit vide. Si ce n'est pas le cas, le thread qui a réalisé cet appel doit être mis en attente.

Un appel `b.get()` renvoie la valeur stockée dans le buffer, seulement si le buffer est plein. Si ce n'est pas le cas, le thread qui a réalisé cet appel doit être mis en attente.

Conditions : exemple (2/2)

Les méthodes `put` et `get` étant exclusives, il est nécessaire d'utiliser un verrou pour les synchroniser.

Cependant, si les conditions pour qu'elles puissent s'exécuter ne sont pas remplies, chacune doit relâcher le verrou pour que l'autre puisse s'exécuter.

Deux solutions sont possibles :

- Chaque méthode doit alors ressortir sans se bloquer et revenir tester l'état du buffer "régulièrement". C'est une attente active (busy wait) très coûteuse.
- Une autre technique est de relâcher le verrou et d'attendre sur une **condition**, de *manière atomique*.

Conditions en Python

Pour permettre de suspendre ou redémarrer un thread selon une certaine condition (représentée par une expression booléenne), Python fournit une classe de verrous de type threading.Condition.

Une condition est utilisée de la même manière qu'un verrou, à l'aide d'une construction `with` ou à l'aide des méthodes `acquire`/`release`.

Les objets de cette classe possèdent également les méthodes `wait`, `notify_all` et `notify` qui permettent de gérer l'attente et le réveil sur une condition.

Étant donnée une condition `c`,

- Un *thread* qui appelle `c.wait()` relâche le verrou de cet objet et, de manière atomique, est mis en attente sur `c`
- Un thread qui appelle la méthode `c.notify_all` réveille **tous** les *threads* en attente sur `c`, tandis que `c.notify()` ne réveille qu'**un seul thread** en attente.

Conditions en Python : exemple

```
cv = threading.Condition()
class Buffer:
    def __init__(self):
        self.full = False
        self.val = None

    def put(self,v):
        with cv:
            while self.full: cv.wait()
            self.val = v
            self.full = True
            cv.notify_all()

    def get(self):
        with cv:
            while not self.full: cv.wait()
            self.full = False
            cv.notify_all()
            return self.val
```

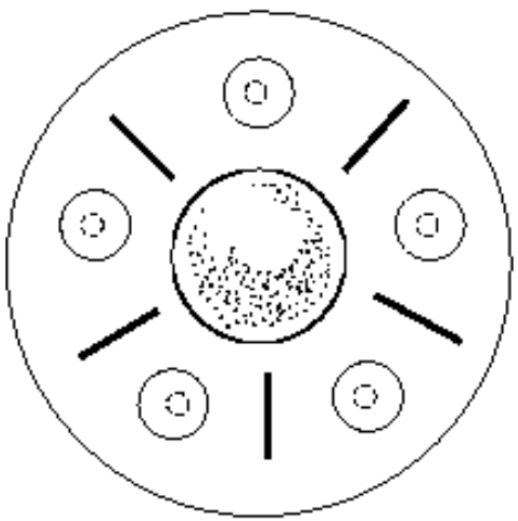
Conditions en Python : remarques importantes

1. Il faut détenir le verrou de la condition pour appeler ses méthodes `wait`, `notify` et `notify_all` ; Autrement, l'exception RuntimeError : cannot notify on un-acquired lock est levée.
2. Les réveils de `notify` et `notify_all` ne sont pas mémorisés.
3. Les *threads* réveillés n'ont aucune priorités
4. On doit itérer sur la condition avec un `while` (et non un `if`) car :
   - un *thread* peut invalider la condition avant que le *thread* réveillé ne reprenne le contrôle
   - le *thread* est réveillé bien que la condition ne soit pas valide (tous les *threads* sont réveillés avec `notify_all`)

Exemple : Les 5 philosophes

Le problème des cinq philosophes est un problème classique de synchronisation proposé par Dijkstra en 1965.

Cinq philosophes assis autour d'une table ronde passent leur temps à réfléchir et à manger. En face de chacun d'eux se trouve un bol de riz et une baguette est placée entre chaque philosophe. Pour manger, chaque philosophe doit prendre les baguettes qui se trouvent autour de lui.



Première solution (1/3)

```
import threading

class Philo(threading.Thread):
    def __init__(self,id,g,d):
        threading.Thread.__init__(self)
        self.g = g
        self.d = d
        self.id = id

    def run(self):
        while True:
            self.g.prendre(self.id)
            self.d.prendre(self.id)
            # le philosophe mange
            self.d.reposer(self.id)
            self.g.reposer(self.id)
```

Première solution (2/3)

```
class Baguette:
    def __init__(self):
      self.cond = threading.Condition()
      self.philo = None

    def prendre(self,p):
      with self.cond:
          self.cond.wait_for(lambda:self.philo==None)
          self.philo = p

    def reposer(self,p):
      if (p==self.philo):
          self.philo = None
          self.cond.notify()

n = int(sys.argv[1])
B = [ Baguette() for i in range(n) ]
P = [ Philo(i,B[i],B[(i+1)%n]) for i in range(n) ]
for i in range(n): P[i].start()
```

Première solution (3/3)

1. Un philosophe attend que sa baguette droite soit disponible
2. Puis, il attend pour obtenir sa baguette gauche
3. Lorsqu'il possède ses deux baguettes, le philosophe mange
4. Après avoir mangé, le philosophe repose ses baguettes et réveille les philosophes en attente.

Première solution (3/3)

1. Un philosophe attend que sa baguette droite soit disponible
2. Puis, il attend pour obtenir sa baguette gauche
3. Lorsqu'il possède ses deux baguettes, le philosophe mange
4. Après avoir mangé, le philosophe repose ses baguettes et réveille les philosophes en attente.

Bien que paraissant évidente, cette solution est fausse, pourquoi ?

Première solution (3/3)

1. Un philosophe attend que sa baguette droite soit disponible
2. Puis, il attend pour obtenir sa baguette gauche
3. Lorsqu'il possède ses deux baguettes, le philosophe mange
4. Après avoir mangé, le philosophe repose ses baguettes et réveille les philosophes en attente.

Bien que paraissant évidente, cette solution est fausse, pourquoi ?

Si tous les philosophes prennent leur baguette droite en même temps, aucun d'eux ne pourra prendre sa baguette gauche, ce qui conduira à un interblocage

Autres solutions

Une autre possibilité est de vérifier, après avoir pris la baguette droite, si la baguette gauche est disponible. Si elle ne l'est pas, le philosophe repose la baguette droite, attend un certain temps et recommence.

Autres solutions

Une autre possibilité est de vérifier, après avoir pris la baguette droite, si la baguette gauche est disponible. Si elle ne l'est pas, le philosophe repose la baguette droite, attend un certain temps et recommence.

Malheureusement cette solution échouera également, pourquoi ?

Autres solutions

Une autre possibilité est de vérifier, après avoir pris la baguette droite, si la baguette gauche est disponible. Si elle ne l'est pas, le philosophe repose la baguette droite, attend un certain temps et recommence.

Malheureusement cette solution échouera également, pourquoi ?

Une amélioration consite à attendre un temps aléatoire si la bagette gauche n'est pas disponible. Bien que la probabilité d'avoir un blocage soit faible, on préfèrera une solution qui marche toujours sans provoquer d'erreur.

Une solution correcte (1/2)

```
import threading
import sys

cond = threading.Condition()

def prendre(p):
    with cond:
        cond.wait_for(lambda: p.g.pense and p.d.pense)
        p.pense = False

def reposer(p):
    with cond:
        p.pense = True
        cond.notify_all()
```

Une solution correcte (2/2)

```
class Philo(threading.Thread):
    def __init__(self,id):
        threading.Thread.__init__(self)
        self.id = id
        self.pense = True

    def run(self):
        while True:
            prendre(self)
            # le philosophe mange
            reposer(self)

n = int(sys.argv[1])
P = [ Philo(i) for i in range(n) ]
for i in range(n):
    P[i].g = P[(i - 1) % n]
    P[i].d = P[(i + 1) % n]
for i in range(n): P[i].start()
```

Sémaphore (1/3)

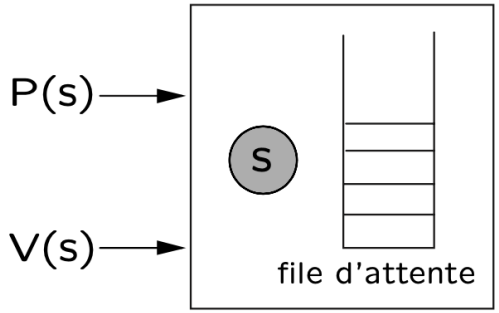
Un sémaphore est une primitive de synchronisation de base définie par Dijkstra en 1968 et utilisée dans le cadre d'un système concurrent avec mémoire partagée

Un sémaphore est défini par :

- Une **structure de données** constituée :
  1. D'une variable *entière* `s`, appelée compteur, ayant initialement une valeur positive ou nulle quelconque fixée par le programmeur
  2. D'une file d'attente `f` initialement vide.
- Deux **opérations**, `P` et `V`, qui peuvent être réalisées sur `s`.

Sémaphore (2/3)

Un sémaphore est une boite noire : Aucune thread ne peut consulter la valeur du compteur et de la file.



Les opérations `P` et `V` sont atomiques. Elles sont définies par :

- `P(s)` teste si `s` > 0. Si le test réussi alors `s` est décrémentée, sinon le *thread* qui a exécuté cette instruction est mis en attente dans la file `f`
- `V(s)` réveille un *thread* en attente dans la file `f` s'il en existe, et incrémente `s` autrement.

L'opération `P(s)` peut donc être bloquante pour le *thread* qui l'exécute, tandis que l'opération `V(s)` n'est jamais bloquante (elle peut par contre réveiller un *thread*)

Sémaphore (3/3)

Cette définition est très générale puisqu'aucun mode de gestion de la file d'attente n'est imposé.

Comment placer un *thread* dans la file et quel *thread* réveiller ?

- Mode aléatoire
- Mode FIFO, "first in, first out" (premier bloqué, premier débloqué)
- Mode LIFO, "last in, last out"
- Mode de gestion basé sur la priorité des *threads*.

Sémaphore en Python

Python fournit une classe `threading.Semaphore` pour créer des sémaphores. Le valeur du compteur initial du sémaphore est donnée lors de l'instanciation.

Un sémaphore initialisé à 1 est équivalent à un verrou.

```
import threading

sem = threading.Semaphore(value=1)

def run():
    while True:
        sem.acquire()
        # section critique
        sem.release()
```

Plus généralement, avec un sémaphore initialisé à k, le nombre maximal de threads en section critique est k.

Solution aux 5 Philosophes avec des sémaphores (1/2)

- Deux philisophes ne peuvent accéder en même temps à une baguette : c'est une section critique
- On utilise un mutex par baguette pour garantir l'exclusion mutuelle.
- Pour empêcher les interblocages, on utilise un sémaphore avec un compteur initialisé à $N - 1$ (où N est le nombre de philosophes).

Solution aux 5 Philosophes avec des sémaphores (2/2)

```
N = int(sys.argv[1])
B = [ threading.Semaphore(1) for i in range(N)]
S = threading.Semaphore(N-1)

def sleep():
    time.sleep(random.uniform(0.5, 1.0))

def run(i):
    while True:
        print(i,': pense')
        S.acquire()
        B[i].acquire()
        B[(i+1)%N].acquire()
        S.release()
        print(i,': mange')
        B[i].release()
        B[(i+1)%N].release()

P = [ threading.Thread(target=run,args=(i,)) for i in
                                range(N) ]
for i in range(N): P[i].start()
```