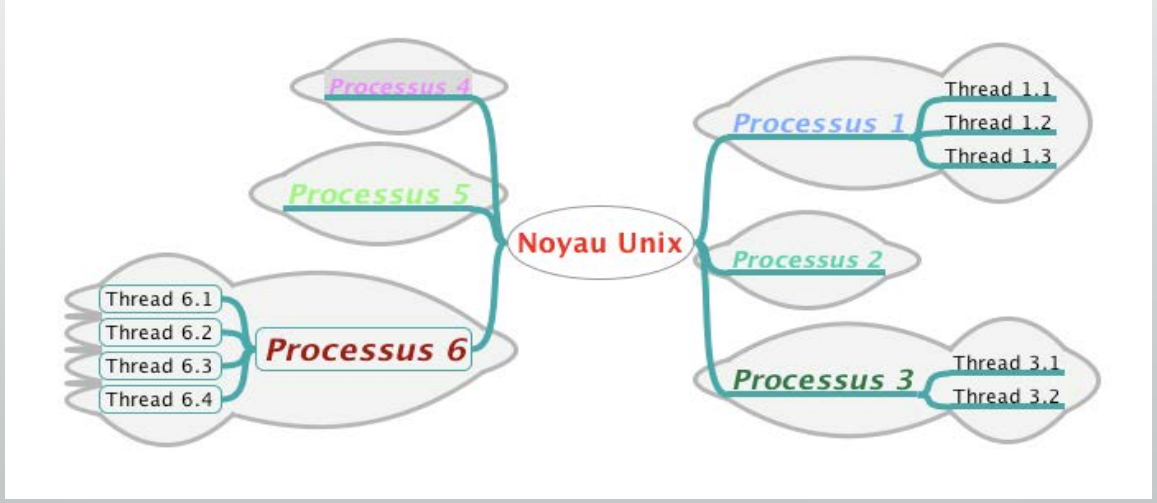
Thread ou processus légers

- Un programme se compose notamment de variables globales et de plusieurs sous-programmes dont le sous-programme principal main
- Un processus lourd se compose initialement d'un processus léger exécutant le sous-programme principal et de ressources (ex : mémoire pour variables globales)
- Le processus lourd au travers de son processus léger initial pourra créer d'autres processus légers qui exécuteront un des sous-programmes du programme
- Les processus légers (le processus initial et ceux créés ultérieurement) s'exécutent en parallèle au sein du processus lourd en **partageant les ressources**



Intérêt des threads

- Certaines applications se dupliquent entièrement au cours du traitement :
  - dans les systèmes client-serveur qui utilisent des processus, le serveur exécute un *fork()* pour traiter chacun de ses clients (un processus par client).
  - Cette duplication est souvent très coûteuse...
- Avec des threads, on peut arriver au même résultat sans gaspillage d'espace, en créant un thread par client, *mais en conservant le même espace d'adressage, de code et de données*. Mais, évidemment, si un client perturbe cet espace, cela impactera tous les clients...
- Les threads sont, par ailleurs, très bien adaptés au parallélisme. Ils peuvent s'exécuter simultanément sur des machines multiprocesseurs et multi-cœurs.

Exemple simple

- f et g mettent à jour deux parties **indépendantes** d'une liste L

  L = [1,2,3,4]

  def f(L)…

  def g(L)…

- On peut paralléliser le traitement, soit :
  - en créant deux processus lourds indépendants L doit être alors copiée dans un **fichier** partagé
  - en créant deux processus légers (threads) indépendants, L est rangée en **mémoire** partagée

```
#processus
pid=os.fork()
if (pid==0):
   f()

pid=fork()
if (pid==0):
   g();
```

```
#threads
thread1 = threading.Thread(target = f, args(L,))
thread2 = threading.Thread(target = g, args(L,))

thread1.start()
thread2.start()
```

Création (En python)

- Le module **threading** fournit une interface permettant de gérer les threads, il fournit:
  - La classe **Thread**
    - La méthode **run** qui implémente le code qui sera exécuté par le thread
    - la méthode **start** qui lance l'exécution du thread
  - un ensemble de fonctions
    - **current_thread()** qui renvoie le thread en cours d'exécution
    - **enumerate()** renvoie la liste des threads en cours d'exécution (y compris le thread principal et courant).
    - **active_count()** renvoie le nombre de threads actifs (égal à la longueur de la liste produite par enumerate()).
    - **main_thread()** renvoie le thread principal (celui qui a lancé tous les autres).
- On peut aussi directement fournir une fonction (et ses paramètres) au constructeur du thread si on n'a pas besoin de créer une sous-classe spécifique.

```
thread1 = threading.Thread(target = f, args(L,))
thread1.start()
```

Terminaison (En python)

- Par défaut, un thread se termine lorsqu'il atteint la fin du bloc de sa méthode run() (ou le corps du programme pour le thread principal).
- Un appel à sys.exit() dans un thread termine celui-ci, pas le programme principal (pratique peu conseillée…)

> ➢ lorsque le thread principal se termine, **tous les threads qu'il a lancé se terminent aussi, même s'ils n'ont pas terminé** : n'oubliez jamais d'appeler les méthodes join() des threads « fils » avant de terminer le thread principal…

Attendre la fin d'un thread

- La méthode d'instance **join()** permet au thread courant d'attendre la fin d'un autre thread. Elle est notamment très importante dans le thread principal car, lorsque celui-ci se termine, *tous les threads qu'il a créé sont supprimés*.
- Un thread t ne peut pas appeler t.join() : cela provoquera une exception *RuntimeError*.

```
for t in threading.enumerate():
   if t != threading.main_thread(): t.join()
```

Threads : Problème

- Soit un système de réservation de billets d'avions. Le principe pourrait être :
  - Trouver une place libre;
  - NbPlacesOccupées = NbPlacesOccupées + 1;
- Ce code peut être exécuté par plusieurs agences de voyages simultanément (processus) : NbPlacesOccupées est donc partagée par tous les processus.
- Deux processus peuvent donc lire quasiment en même temps la valeur de NbPlacesOccupées et le résultat de ces deux réservations n'occupera donc qu'une seule place de plus...

Code python

```
import threading

nb_places_occupees = 0

def inc():
    global nb_places_occupees
    for i in range(100000):        # incrémente 100000 la var partagée
        nb_places_occupees += 1

# Thread principal
for i in range(10): # Création et lancement de 10 threads
    threading.Thread(target=inc).start()

for t in threading.enumerate():
    if t != threading.main_thread(): t.join()

print("Nbre de places occupées = {}".format(nb_places_occupees))
```

Le problème vient du fait que plusieurs threads lisent et modifient simultanément une variable partagée.

Accès à des objets partagés

- *Deux processus/threads ne doivent jamais pouvoir manipuler simultanément une variable commune* : il faut garantir *l'accès en exclusion mutuelle* aux variables partagées.
- On appellera ***section critique*** un segment de code qui manipule des variables globales.
- Pour que l'accès à ces variables se fasse en exclusion mutuelle, on ajoutera du code :
  - Un *prologue* pour attendre d'entrer en section critique.
  - Un *épilogue* pour signaler qu'on en est sorti
- Les threads/processus pourront ainsi arbitrer l'accès à la section critique

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
  # Section critique
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
```

Construisons une solution valide

- Contraintes
  - À tout instant, un processus au plus est engagé en SC.
  - Aucune hypothèse ne sera faite sur la vitesse des processus.
  - Tout processus peut être interrompu à tout instant.
  - Pas de famine.
  - Si la SC est libre, tout processus demandant à y entrer doit pouvoir le faire (pas de "tour de rôle").

1ère Tentative

- utilisation d'un booléen LIBRE initialisé à VRAI pour indiquer que la SC est libre ou non
  - LIBRE doit donc être une variable globale, partagée par tous les processus…

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
while (NOT LIBRE)    # Attente active
LIBRE = FAUX         # Entrée en SC
# Section critique
SC                   # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
LIBRE = VRAI         # Sortie de SC
```

1ère Tentative : Analyse

- Si deux processus veulent entrer en même temps en SC et que LIBRE est à VRAI, ils vont y entrer tous les deux en même temps en mettant tous les deux LIBRE à VRAI en ressortant.
  - Supposons que P0 trouve LIBRE à VRAI et qu'une interruption survienne avant qu'il ait eu le temps de la mettre à FAUX.
  - Supposons que cette interruption donne le contrôle à P1 qui, lui aussi, veut entrer en SC : il peut le faire.
  - Quand P0 reprend le contrôle, il reprend ou il s'était arrêté : il met LIBRE à FAUX et entre en SC → **Cette solution n'est pas bonne.**

On ne peut pas trouver de solution correcte avec un seul booléen. De plus, le fait que LIBRE soit une variable commune n'a fait que déplacer le problème : maintenant, c'est LIBRE qui est le sujet de l'exclusion mutuelle !

2ème Tentative

- Utilisation d'un entier TOUR donnant le numéro du processus dont c'est le tour de passer.
- On suppose que chaque processus connaît son numéro à l'aide d'une constante MOI (qui vaut 0 ou 1).
- Initialement, TOUR est initialisé à 0.

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
while (TOUR != MOI)  # Attente active
TOUR = MOI           # Entrée en SC
# Section critique
SC                   # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
TOUR = 1-MOI         # Sortie de SC
```

2ème Tentative : Analyse

- Si P0 et P1 sont tous les deux en SC :
  - Si P0 est entré, c'est qu'il a trouvé TOUR à 0.
  - Si P1 est entré, c'est qu'il a trouvé TOUR à 1.
  - Ce qui est absurde…

- Exclusion mutuelle GAGNEE mais alternance imposée entre P0 et P1

  ☹ Si la SC est libre, tout processus demandant à y entrer doit pouvoir le faire (pas de "tour de rôle"). → Contraintes non respectées

3ème Tentative

- On utilise un tableau global de booléens IN, initialisé à FAUX et deux variables locales aux processus, MOI (qui vaut 0 ou 1) et TOI (qui vaut 1 – MOI)

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
while (IN[TOI])      # Attente active
IN[MOI] = VRAI       # Entrée en SC
# Section critique
SC                   # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
IN[MOI] = FAUX       # Sortie de SC
```

3ème Tentative : Analyse

- Si deux processus veulent entrer en même temps en SC et qu'elle est libre, IN vaudra alors FAUX,FAUX et les deux processus pourront entrer en même temps,

  ☹ il n'y a donc pas exclusion mutuelle…

4ème Tentative

- On utilise un tableau REQUETE qui désigne les demandes (initialisé à FAUX).

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
REQUETE[MOI] = VRAI
while (REQUETE[TOI]) # Attente active
# Section critique
SC                   # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
REQUETE [MOI] = FAUX   # Sortie de SC
```

4ème Tentative : Analyse

☺Exclusion mutuelle ok

☹Mais Si deux processus positionnent en même temps leur REQUEST[MOI] et qu'ils testent le REQUEST[TOI], chacun boucle indéfiniment. On a donc une **situation d'interblocage**

Algorithme de Peterson (1981)

- Mélange des tentatives précédentes qui utilise
  - un tableau de booléens (initialisé à FAUX)
  - et un entier comme variables globales

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
REQUETE[MOI] = VRAI
TOUR = 1-MOI
while (REQUETE[TOI] et TOUR ==(1-MOI))    # Attente active
# Section critique
SC                    # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
REQUETE [MOI] = FAUX   # Sortie de SC
```

Les sémaphores de Dijkstra

- Un sémaphore est un objet abstrait sur lequel on peut réaliser deux opérations atomiques
  - P(Sem) (prolagen (essayer de diminuer))
  - V(Sem) (verhogen (augmenter))
- P est l'opération susceptible de bloquer, V n'est jamais bloquant mais débloque un processus bloqué par P.
- Un sémaphore étant un objet commun à plusieurs processus, toutes les opérations P et V agissant sur un même processus doivent se faire en exclusion mutuelle.
- À un sémaphore S est associé, de façon interne :
  - un compteur cpt, qui est un entier représentant la valeur du sémaphore
  - une file d'attente f, initialement vide.

Les sémaphores : Fonctionnement

- P(Sem)

```
S.cpt = S.cpt - 1;
if (S.cpt < 0) {
   Insérer le processus exécutant P dans S.f ;
   Bloquer ce processus ;
}
```

- V(Sem)

```
S.cpt = S.cpt + 1;
if (S.cpt <= 0) { /* il y a des processus bloqués */
   Extraire le processus en tête de S.f;
   Débloquer ce processus ;
}
```

Les sémaphores : Remarques

- Le nombre de processus bloqués par un sémaphore S est égal à S.cpt
- Un processus bloqué par P(S) devra attendre qu'un autre processus exécute V(S).
- Pour initialiser le sémaphore, on utilise généralement un appel comme Sem_Init(S, val) (ou on utilise son constructeur s'il est implémenté comme une classe...)
- Un sémaphore permet donc de contrôler le nombre de processus autorisés à entrer dans une section critique.
- Un *MUTEX* est un sémaphore initialisé à 1, qui permet donc d'assurer l'exclusion mutuelle.

Les sémaphores : En python

- Le module **threading** fournit une classe **Semaphore** permettant de créer des sémaphores d'exclusion mutuelle entre les threads. Cette classe implémente l'opération P via une méthode **acquire()** et l'opération V via une méthode **release()**.
- Le constructeur prend une valeur initiale pour le compteur. Par défaut, cette valeur est 1 (ce qui convient donc pour un mutex).
- Pour garantir que release() soit toujours appelée (quoi qui se passe dans la section critique), le plus sûr est d'utiliser un bloc

try : ...

finally : ... :

```
mutex.acquire() # P(mutex)
try:
  code de la section critique
finally:
  mutex.release() # V(mutex)
```

Réservation Avion : Solution

```
import threading

mutex = threading.Semaphore() # Ou threading.Lock() – voir plus loin
nb_places_occupees = 0

class Inc(threading.Thread):
   def run(self):
      global nb_places_occupees
      for i in range(100000):
           mutex.acquire()
           try:
              nb_places_occupees += 1
           finally:
              mutex.release()

# Thread principal
for i in range(10):
   Inc().start() # Lancement de 10 threads
for t in threading.enumerate():
   if t != threading.main_thread(): t.join()
print("Nbre de places occupées = {}".format(nb_places_occupees)) # 1000000...
```

```
with mutex:
   nb_places_occupees += 1
```

Les sémaphores : Utilisation

- La classe Semaphore est principalement utilisée pour limiter l'accès à une ressource dont les capacités sont limitées (autoriser un nombre maximum d'accès)
  - Dans la réservation de places d'avions, l'accès à la variable nb_place_occupees est limitée à une seule personne → la classe **threading.Lock** suffit amplement.
- Un sémaphore ou un verrou doivent être vus comme un moyen de contrôler l'accès à une zone de code → ce n'est pas un moyen de communiquer entre thread...
- Pour synchroniser plusieurs threads entre eux, il existe d'autres mécanismes : les variables conditions ou les files synchronisées, par exemple.

Les sémaphores : mauvais exemple

```
import threading

mutex_ping = threading.Semaphore()
mutex_pong = threading.Semaphore(0) # on bloque les pong dès le départ

class Ping(threading.Thread):
   def run(self):
      for i in range(100): # Affiche 100 ping
           mutex_ping.acquire()
           print("Ping...", end=" ")
           mutex_pong.release()

class Pong(threading.Thread):
   def run(self):
      for i in range(100): # Affiche 100 pong
           mutex_pong.acquire()
           print("Pong")
           mutex_ping.release()

# Prog principal
Pong().start()
Ping().start()
for t in threading.enumerate():
   if t != threading.main_thread(): t.join()
```

Exercice : La voie unique

On considère des véhicules circulant sur des voies parallèles, mais avec parfois des tronçons à voie unique. Bien entendu, il ne peut y avoir deux véhicules circulant en sens inverse sur une telle voie unique.

1. Par quoi est caractérisé un véhicule
2. Donner le comportement général d'un véhicule circulant sur ces voies
3. Cas 1 : Il ne peut y avoir qu'un seul véhicule sur la voie unique
    1. Identifier la section critique
    2. Comment la protéger

- Cas 2 : Le nombre de véhicules à un instant donné sur la voie unique est illimité. La solution proposée ne doit pas imposer l'alternance des passages

```
void Processus Vehicule (int monSens) {
        Circuler sur portion à double sens ;
        DemanderAccesVU (monSens) ;
        Circuler sur la voie à sens unique ;
        LibererAccesVU() ;
        Continuer à circuler sur la voie à double sens ;
}
```