# ­TP7 (3h) : Accès au fichiers avec mmap ; Partage de mémoire entre processus ; Synchronisation entre processus via sémapohores

**Rappel : utiliser la version 3 de Python (commande python3).**

**Il est recommandé de commencer vos sources par la ligne suivante :**

*#! /usr/bin/env python3*

**Rappel sur la documentation.** Les fonctions du module **os** en Python sont des copies conformes des fonctions système en langage C, qui disposent de pages manuel UNIX (**man**). La documentation du module Python **os** étant souvent assez succincte, il est utile de consulter également les pages **man** des fonctions C utilisées.

1. **Accès en lecture/écriture avec mmap**

*La fonction POSIX* ***mmap*** *permet un autre type d’accès à un fichier : une partie du fichier synchronisée avec une zone mémoire, les lectures/écritures dans le fichier pouvant se faire comme de simples accès mémoire via un pointeur (en C) respectivement un tableau de* bytes *(en Python).*

Ecrire un programme qui prend en argument le nom d’un fichier binaire comportant une série de entiers sur 16 bits représentés en *little-endian* et le convertit en *big-endian*  (cf. <https://fr.wikipedia.org/wiki/Endianness> )

A noter :

1. le fichier sera mappé en mémoire avec **mmap**
2. la conversion se fera par un simple parcours du tableau *bytes* résultant du **mmap**, en intervertissant les paires d’octets successifs.
3. un fichier de test est fourni. Vous pouvez examiner son contenu avec la commande **od --format=x1 test.bin**
4. **Communication entre processus via un segment de mémoire partagée**

*Le module* ***posix\_ipc*** *donne accès à différents mécanismes POSIX de communication entre processus depuis Python, cf.* [*http://semanchuk.com/philip/posix\_ipc*](http://semanchuk.com/philip/posix_ipc)*. La classe* ***SharedMemory***  *permet de créer des segments de mémoire partagés entre processus (elle fait appel à la fonction système* ***shm\_open****). Ces segments doivent ensuite être mappés en mémoire avec la fonction* ***mmap****. Le contenu est ensuite accessible sous forme de tableau de bytes, comme à l’exercice précédent.*

*A noter aussi : si* ***tab*** *est un tableau de bytes, on peut utiliser les* ***slices*** *python pour accéder à son contenu. Par exemple :*

* ***a[start:end]*** *désigne les items de* ***start*** *à* ***end-1***
* ***a[start:]*** *désigne les items de* ***start*** *à la fin du tableau*
* ***a[:end]*** *désigne les items du début à* ***end-1***
* *les indices négatifs sont possibles et sont considérés relatifs à la fin du tableau*

*Un* ***slice*** *peut être utilisé aussi bien du côté droit que du côté gauche d’une affectation (pour lire ou pour écrire dans le tableau).*

L’objectif de l’exercice est d’écrire deux programmes (P1,P2) qui communiquent un tableau d’entiers (codés chacun sur 1 octet) par mémoire partagée. Les deux programmes commencent par :

1. Créer/accéder la mémoire partagée en utilisant un nom unique dans le constructeur de **SharedMemory** (attention, le nom doit commencer par un "/").
2. Mapper le segment partagé avec **mmap** pour pouvoir y accéder sous forme de tableau de bytes.

Le fonctionnement par la suite est le suivant :

* P1 :
  + Saisie de la taille et du contenu d’un tableau d’entiers (valeurs entre 0 et 255)
  + Stockage de la taille du tableau et du contenu dans la mémoire partagée
  + Mise en attente pour 30s (**sleep**)
  + Fermeture du **mmap** (avec la méthode **close**)
  + Suppression de la mémoire partagée (avec la méthode **unlink**)
* P2 :
  + Affichage de la taille et du contenu du tableau à partir de la mémoire partagée
  + Fermeture du **mmap**

Pour que les deux programmes puissent fonctionner correctement, il faut lancer P1 en premier, puis P2 dans un autre terminal pendant les 30s d’attente de P1.

Question : lister le contenu du répertoire **/dev/shm** *pendant* l’exécution de P1. Que concluez-vous ?

1. **Synchronisation de processus de par sémaphore**

*Cet exercice est la suite de l’exercice précèdent.*

Les 30s de **sleep** de P1 servent à permettre à l’utilisateur le lancement de P2 pendant cet intervalle. Ceci n’est pas une bonne solution pour la synchronisation. On voudrait que P2 puisse être lancé en premier et qu’il attende que les données soient prêtes avant de les utiliser. Ceci peut se faire en utilisant un *sémaphore* pour synchroniser P1 et P2.

Le module **posix\_ipc** permet d’utiliser les sémaphores POSIX du système, cf. documentation : <http://semanchuk.com/philip/posix_ipc/#semaphore>

Modifier les programmes P1 et P2 pour qu’ils allouent un sémaphore et l’utilisent pour la synchronisation indiquée. A la fin du programme P2, on devra libérer le sémaphore.  
L’opération **unlink** de P1 risque de poser un petit problème à P2 pour accéder aux données, lequel ? Proposez une solution.

1. **BONUS : mémoire partagée – fonctions System V en langage C**

*Un autre API (System V) pour communiquer via mémoire partagée est largement répandu et utilisé. Il est décrit dans le chapitre 5 de* Advanced Linux Programming*. Se référer aux pages de man des fonctions ci-dessous.*

Créer une version C des programmes P1, P2 de l’exercice précédent : les deux programmes commencent par créer/accéder la mémoire partagée (**shmget**) à partir d’une même clé, générée à partir du chemin du répertoire de travail (fonctions **ftok** et **getcwd**). *Le répertoire de travail à l’exécution devra donc être le même pour les deux processus.* Attacher le segment dans l’espace mémoire avec **shmat**.

Le fonctionnement par la suite est le suivant :

* P1 :
  + Saisie/stockage du tableau d’entiers dans la mémoire partagée
  + Se met en attente pour 15s (**sleep**)
  + Détache la mémoire partagée (**shmdt)**
* P2 :
  + Affiche le contenu du tableau à partir de la mémoire partagée
  + Détache et libère la mémoire partagée (**shmdt**, puis **shmctl** avec commande IPC\_RMID)

Questions:

1. Entre le moment où P2 se termine et la fin de P1, dans un autre terminal, lancez la commande **ipcs** (vous avez 15s !). A la fin de P1 lancez-la de nouveau. Que pouvez-vous conclure ?
2. Dans P1, inversez l’ordre de **sleep** et **shmdt**. Répétez le point a). Quelle conclusion maintenant ?
3. Toujours dans P1, ajoutez à la fin (après le **sleep**) un nouvel appel à **shmat** pour essayer d’attacher la mémoire à nouveau, avec traitement d’erreurs (perror). Que se passe-t-il ?

A la fin de l’exercice, vérifiez avec **ipcs** s’il reste des segments non-libérés et les libérer avec la commande **ipcrm**.