# TP 2 – Threads, signaux et sémaphores

## Exercice 1

On souhaite réaliser un programme composé de $n$ nouveaux threads utilisant des signaux. L'objectif est que chaque nouvelle pression sur CTRL-C (correspondant à chaque fois à la génération du signal `SIGINT`) provoque la terminaison du thread numéro $i$ ($0 \leq i < n$).

Comme on ne sait pas à priori à quel thread va être remis le signal `SIGINT`, les $n$ nouveaux threads vont devoir le masquer (avec `pthread_sigmask`) de telle sorte que seul le thread principal puisse le recevoir. Celui-ci va alors signaler au thread $i$, via l'envoi du signal `SIGUSR1`, qu'il doit se terminer.

On demande donc de réaliser le programme de la manière suivante :

1. le thread principal positionne les fonctions associés aux signaux `SIGINT` et `SIGUSR1` grâce à la primitive `sigaction` (on rappelle que l'action associée à chaque signal est globale à l'ensemble des threads d'un processus) ;
2. puis, il génère les $n$ threads ;
3. chaque nouveau thread doit masquer la réception du signal `SIGINT` ;
4. les $n + 1$ threads (y compris le thread principal) se synchronisent alors avec une barrière de façon qu'aucun thread ne fasse une action avant que tous soient prêts ;
5. une fois la barrière ouverte, chaque thread attend l'arrivée d'un signal avec la primitive `pause` ;
6. le thread principal, quant à lui, prend en compte l'arrivée de `SIGINT` ; à chaque réception de `SIGINT`, il envoie `SIGUSR1` au suivant des nouveaux threads, puis il en attend la terminaison ;
7. à la réception de `SIGUSR1`, le thread numéro $i$ se termine ;
8. le thread principal se termine lorsque les $n$ nouveaux threads sont terminés.

Dans cet exercice, vous n'utiliserez aucune variable globale.

## Exercice 2

Reprendre les programmes `producteur` et `consommateur` du TD 2, et remplacer le système de temporisations par une synchronisation à base de sémaphores IPC System V. Vous utiliserez un seul groupe de sémaphores. Les prototypes de certaines de vos fonctions pourront être modifiés.

Pour mémoire, le texte du TD était :

> On souhaite faire communiquer deux programmes via un segment de mémoire partagée :
> — le premier programme, qu'on nommera `consommateur`, admet en argument une valeur $n$, crée le segment de mémoire partagée (en le supprimant au préalable s'il existait) de taille suffisante pour stocker $n$ entiers strictement positifs (en plus des indices mentionnés ci-dessous), puis attend des valeurs dans la mémoire partagée pour les imprimer sur la sortie standard. Lorsque le programme lit la valeur 0, il détruit le segment de mémoire partagée et se termine.
> Après chaque lecture dans le segment de mémoire partagée (lecture d'un entier ou absence d'entier), le consommateur se met en attente pendant une seconde.
> — le deuxième programme, qu'on nommera `producteur`, prend un argument obligatoire, une valeur positive (éventuellement nulle pour arrêter le consommateur) à écrire dans le segment de mémoire partagée.
>
> Ces programmes demandent une organisation de la mémoire partagée. Pour cela, lors de l'initialisation, le consommateur place au début du segment 3 emplacements (en plus des $n$ entiers demandés) pour contenir les valeurs suivantes :
> — nombre maximum d'entiers ($n$) dans le segment
> — indice de l'entier suivant à lire (initialisé à 0)
> — indice de l'entier suivant à écrire (initialisé à 0)

Pour réaliser la lecture d'un entier, le consommateur compare l'indice de l'entier suivant à lire avec l'indice de l'entier suivant à écrire. S'il y a au moins un entier, il est lu et l'indice correspondant est mis à jour. Pour réaliser l'écriture d'un entier, le producteur fait de même. S'il n'y a plus de place, le producteur se met en attente pendant une seconde, puis recommence.

Dans ce qui suit, on se contentera des temporisations et on ne cherchera pas à placer de synchronisations pour rendre les programmes rigoureux.

Pour réaliser le consommateur, on demande de programmer les fonctions suivantes :

1. `int creer_shm (int n)` : supprime le segment de mémoire partagée s'il existe, puis le crée avec la taille nécessaire pour les $n$ entiers, et retourne l'identificateur de segment (valeur de retour de `shmget`) ;
2. `void supprimer_shm (int shmid)` : supprime le segment de mémoire partagée, pour utilisation en fin de programme ;
3. `int lire_entier (void *adr)` : lit un entier et retourne sa valeur (pouvant être 0) ou -1 si aucun entier n'est disponible dans le segment. L'adresse est celle du segment de mémoire partagée (vous pouvez utiliser un autre type à la place de `void` si cela vous arrange) ;
4. écrivez à présent la fonction `main` qui analyse l'argument, crée, attache et initialise le segment de mémoire partagée, puis lit des entiers en se mettant en attente après chaque lecture (réussie ou non). Lorsque la valeur 0 est lue, le programme supprime le segment de mémoire partagée et se termine.

Pour réaliser le producteur, on demande de programmer les fonctions suivantes :

5. `void *acceder_segment (void)` : initialise l'accès au segment de mémoire partagée et retourne l'adresse du segment de mémoire partagée (vous pouvez utiliser un autre type que `void` si cela vous arrange) ;
6. `int ecrire_entier (void *adr, int val)` : écrit l'entier et retourne 1 s'il a pu être écrit, ou 0 s'il n'y a plus de place dans le segment. L'adresse est celle du segment de mémoire partagée (vous pouvez utiliser un autre type à la place de `void` si cela vous arrange) ;
7. écrivez à présent la fonction `main` qui analyse l'argument, attache le segment de mémoire partagée, puis écrit l'entier en bouclant avec une attente s'il n'y a plus de place dans le segment. Lorsque l'entier est finalement écrit, le programme se termine.