Nous allons découvrir principalement dans ce TP l'usage des appels systèmes `fork` et `pipe`. Ces deux appels systèmes vous permettront d'aborder la programmation concurrente, c'est-à-dire la création d'un programme s'exécutant sur plusieurs processus distincts et donc en parallèle. Il est à noter que nous n'aborderons pas les processus légers appelés aussi *threads*.

Le Fork épique

La fonction `fork` permet de dupliquer le processus courant. Cette duplication inclut les valeurs en mémoire, les descripteurs de fichier et la position dans le code en cours d'exécution. Ainsi, un fichier ouvert grâce à un appel à la fonction `open` sera aussi ouvert dans le nouveau processus. Nous qualifierons de *processus fils* le processus nouvellement créé et de *processus père* le processus ayant fait l'appel à la fonction `fork`. Le prototype de la fonction `fork` est le suivant

```
pid_t fork(void);
```

Nous pouvons voir que la fonction `fork` retourne une valeur. Cette valeur est différente suivant si l'on se trouve dans le processus père ou dans le processus fils au retour de l'appel. En effet, le processus père et le processus fils étant identiques, il nous faut les distinguer si nous voulons exécuter deux codes différents. Ceci est possible grâce à la valeur retournée par la fonction `fork`. Dans le processus fils, la fonction `fork` retourne `0`. Dans le processus père, la fonction `fork` retourne le *pid* du processus fils venant d'être créé.

Un pid est un numéro unique identifiant un processus. Il est à noter que la valeur `0` n'est pas un pid valide et que la valeur maximum que peut prendre un pid est disponible dans le fichier `/proc/sys/kernel/pid_max`. De plus, lorsqu'un processus se termine, son pid est libéré pour une utilisation ultérieure.

Q 1. Écrire un programme qui créera un processus fils. Chaque processus affichera ensuite si il est le processus père ou le processus fils accompagné de son pid respectif. Vous utiliserez la fonction `getpid` pour obtenir le pid du processus — faire `man getpid` pour plus de détails.

Q 2. Reprenez le programme précédent et après l'affichage, faites dormir le processus père pendant 1 seconde et le processus fils pendant 5 secondes puis faites leur afficher un message précisant qui ils sont. Vous utiliserez la fonction `sleep` pour faire dormir un processus — faire `man 3 sleep` pour plus de détails. Qu'observez vous ? Lancez votre programme en tâche de fond — ajouter `&` après votre programme — et exécutez la commande `ps` à plusieurs reprises. Qu'observez vous ? Inversez les durées de sommeil des deux processus et trouvez la différence.

Q 3. Reprenez le programme précédent et faites afficher aux deux processus après le retour de la fonction `sleep` le pid de leur père en utilisant la fonction `getppid` — faites `man getppid` pour plus de détails. Qu'observez vous ? Inversez les durées de sommeils des deux processus et trouvez la différence.

Un processus va traverser au cours de sa vie différents états, les plus important étant les états *actif*, *endormi*, *suspendu* et *zombi*. Dans les questions précédentes, vous avez dû rencontrer l'état zombie. Un processus dans cet état est un processus qui s'est terminé mais dont le père n'a pas encore pris connaissance de sa terminaison. Si le père d'un processus se termine avant ses fils, ceux-ci sont rattachés au processus de pid 1 qui est *init*.

Pour éliminer les processus fils zombi d'un processus père, les fonctions `wait` et `waitpid` peuvent être utilisées. Toutes deux permettent d'attendre le passage d'un des processus fils dans l'état zombi et retournent son pid. Si le processus père ne possède pas de processus fils, `-1` est retourné sinon l'appel est bloquant. Le caractère bloquant peut être supprimé pour la fonction `waitpid` en passant l'option `WNOHANG`. Faites `man wait` ou `man waitpid` pour plus de détails.

Ainsi un processus père désire attendre la terminaison de tous ses fils, il bouclera sur un appel à `wait` jusqu'à ce que celui-ci lui retourne `-1`.

Q 4. Reprenez le programme précédent et indépendamment des durées de sommeil, faites en sorte que le processus père se termine après le processus fils.

Q 5. Reprenez le programme précédent et faites en sorte que le père crée deux fils et non plus un seul. On fera attention à ce que le premier fils créé ne crée pas lui même de fils.

Le Pipe de la communication

Nous avons vu comment créer de nouveau processus et comment synchroniser leurs terminaisons, nous allons voir maintenant comment les faire communiquer ensemble. Une solution intuitive pour faire communiquer deux processus distincts serait de les faire écrire dans un fichier dont l'emplacement est connu à l'avance. Cependant, cette solution n'est pas viable en terme de sécurité et de fiabilité. Un appel système existe pour créer un tuyau de communication, c'est la fonction `pipe`. Le prototype de cette fonction est le suivant

```
int pipe(int pipefd[2]);
```

Cette fonction prend en paramètre un tableau de deux entiers qui au retour de l'appel contiendra deux descripteurs de fichier. Le descripteur de fichier à l'indice 0 du tableau est pour lire dans le tuyau (pipe) et le descripteur de fichier à l'indice 1 du tableau est pour écrire dans le tuyau. Il est à noter que la lecture est destructive.

Pour lire et écrire dans un pipe, on utilisera les fonctions vues précédemment, `read` et `write`. Il est à noter que la lecture est bloquante si il n'y pas de données disponibles et que l'écriture est bloquante si le pipe est plein.

Pour distribuer des tâches sur plusieurs processus, une organisation fréquemment utilisée est celle de maître-esclaves. Dans cette organisation, un processus jouera le rôle de processus maître et distribuera les tâches au processus esclaves ceux-ci envoyant éventuellement un résultat au processus maître. Nous allons voir maintenant comment mettre en pratique cette organisation de différentes manières.

Q 6. Écrivez un programme qui crée deux processus fils, lit sur l'entrée standard des entiers entrés au clavier — typiquement en utilisant `scanf` — et envoie l'entier à l'un de ces fils — alternativement ou de manière aléatoire. Les fils feront la somme des entiers qu'ils recevront et lorsqu'il n'y aura plus d'entier à lire, afficheront la somme ainsi calculée. Vous devez ici vous poser plusieurs questions : Quand créer un pipe et combien ? Comment envoyer un entier au travers d'un pipe ? Comment lire un entier au travers d'un pipe ? Comment savoir qu'il n'y aura plus de données à lire sur le pipe ?

Q 7. Reprenez le programme précédent et modifiez pour que les processus fils envoient au processus père la somme qu'ils ont calculés. Le processus père affichera les deux sommes. Est-ce que le père doit attendre la terminaison de ses deux fils ?

Q 8. Reprenez le programme précédent et minimisez le nombre de pipe utilisés. Comment faire si le père doit savoir de quel fils vient le résultat ?

Mise en pratique : le jeu du chaos - génération d'un triangle de Sierpiński

Le but du programme est de générer un triangle de Sierpiński en utilisant la méthode du jeu du chaos. Le résultat de la génération sera affiché sur l'écran grâce à la fonction `display` fourni. Celle-ci lira les points à afficher sur le descripteur de fichier passé en paramètre. Le programme `game_chaos` que nous allons écrire prendra en paramètre un argument : le nombre de points à générer par triangles.

```
Usage: chaos_game nbPoints
```

Le programme créera un processus fils appelant la fonction `display` avec l'extrémité en lecture d'un pipe en paramètre. Le programme devra ensuite lancer un processus par triangle pour générer les points et les écrire sur le pipe. Enfin le programme attendra la fin du programme `viewer` avant de se terminer.

La fonction `main` vous est donnée dans le fichier `chaos_game.c`. Les structures de données utilisées pour décrire un triangle et un point sont données dans le fichier `chaos_game.c`. Le fichier `viewer.c` contient la fonction `display`. Un `makefile` vous est fourni, il est fortement recommandé de l'utiliser — surtout à cause de l'utilisation de la librairie SDL.

Q 9. Écrire la fonction

```
int createViewer(int width, int height, int* writeToViewerFd);
```

qui crée un pipe, range l'extrémité en écriture dans la variable `writeToViewerFd` et crée un nouveau processus qui appellera la fonction `display` avec l'extrémité en lecture du pipe en paramètre. On fera attention à fermer les extrémités inutiles du pipe. La fonction retournera `-1` en cas d'erreur, le pid du processus créé sinon.

Q 10. Écrire la fonction

```
int sendPointToFd(int fd, point p);
```

qui écrit le point `p` sur le descripteur de fichier `fd`. La fonction retournera `-1` en cas d'erreur, `0` sinon.

Q 11. Écrire la fonction

```
int generateSierpinskiTriangle(int writeToViewerFd, triangle t, point seed, unsigned int nbPoints);
```

qui crée un processus fils dans lequel les points générés sont écrit sur le descripteur de fichier `writeToViewerFd`. La génération des points est effectuée en utilisant la fonction `getNewPoint`. On appellera initialement cette fonction avec le point `seed` en second paramètre puis pour les appels suivant le point retourné précédemment par la fonction. On n'écrira pas sur le descripteur fichier `writeToViewerFd` les 10 premiers points générés. Une fois `nbPoints` points générés, le processus se terminera. La fonction retournera `-1` en cas d'erreurs, le pid du processus créé sinon. On utilisera la fonction `sendPointToFd` pour écrire un point sur le descripteur de fichier.

Q 12. Écrire la fonction

```
void cleanUpZombies(int groupId);
```

qui supprime les processus zombies appartenant au groupe de processus `groupId`.

Q 13. Écrire la fonction

```
void waitingForTheEnd(int pidViewer);
```

qui attend la fin du processus de pid `pidViewer` pour retourner.