TP 5 – *Pipes* et processus

Vous devez faire au moins les exercices 1 à 3.
Un exemple de sujet d'examen concernant cette partie est donné dans l'exercice 4.
L'exercice 5 est un complément, à faire uniquement si vous avez fini tous les autres exercices des différents sujets de TP.

Exercice 1 - flux, fichiers réguliers et fifo

Cet exercice porte sur le programme `mycat`, dont les sources sont disponibles dans le fichier `/pub/FISE_OSSE11/syscall/mycat.c`.

Ce programme met en évidence le comportement des appels système

```
ssize_t read(int fd, void* buf, size_t count)
```

et

```
ssize_t write(int fd, const void* buf, size_t count)
```

en fonction du type de fichier.

La table ci-dessous résume (en supposant que les arguments sont valides) le comportement attendu :

	mode	bloquant	retour		SIGPIPE
			0	-1	
read	regulier	jamais	fin fichier	non	jamais
write	regulier	jamais	jamais	full	jamais
read	fifo	V & $ne \neq 0$	V & $ne = 0$	jamais	jamais
write	fifo	P & $nl \neq 0$	jamais	spa & $nl = 0$	$nl = 0$

ne = nombre d'écrivains — V = FIFO vide
nl = nombre de lecteurs — P = FIFO pleine
$full$ = disque plein ou dépassement de quota
spa = signal `SIGPIPE` attrapé ou ignoré

1. Décrivez le comportement du programme `mycat` :

   lignes 24 + 10 à 15
   Attache le ..... au signal `SIGPIPE`.
   Le gestionnaire affiche un message, attend ... secondes, puis termine le processus avec 0 comme code de retour.

   lignes 26 à 38 boucle infinie :
   - **ligne 28** Lecture d'un caractère `c` sur le flux ..... ;
   - **ligne 29 à 31** Si une ..... de lecture s'est produite, on écrit un message d'erreur et on ..... le processus ;
   - **ligne 32 à 34** Si la .... du flux d'entrée est détectée, on écrit un ..... sur le flux ....., puis on ..... 0.1 seconde ;
   - **ligne 36** Dans les autres cas, on écrit le caractère `c` sur le flux ..... .

2. Compilez le fichier `/pub/FISE_OSSE11/syscall/mycat.c`.

```
sh$ gcc -Wall -Wextra -o mycat /pub/FISE_OSSE11/syscall/mycat.c
```

3. Expérimentation de `mycat` sur des flux réguliers.

   a. Dans quatre terminaux différents, lancez deux écrivains sur le fichier `file` et deux lecteurs sur ce même fichier.

```
sh$ xterm -e bash -c "./mycat > file" &
sh$ xterm -e bash -c "./mycat > file" &
sh$ xterm -e bash -c "./mycat < file" &
sh$ xterm -e bash -c "./mycat < file" &
```

b. Tapez les entrées suivantes dans les fenêtres des écrivains, en regardant leurs résultats dans les fenêtres des lecteurs.

aaaa<ENTER>bbbb<ENTER>	dans la fenêtre du premier écrivain.
cccc<ENTER>dddd<ENTER>eeee<ENTER>	dans la fenêtre du deuxième écrivain.
ffff<ENTER>gggg<ENTER>	dans la fenêtre du premier écrivain.

c. Stoppez les processus lecteurs avec <CTRL-S> (vous pourrez les réveiller avec <CTRL-Q>).

d. Soit P_1 et P_2 deux processus, et f fichier régulier :

— Une lecture sur f n'est pas bloquante. Si on est en fin de fichier, le noyau renvoie une valeur indiquant la fin de fichier.

— Si P_1 et P_2 écrivent dans f à la même position, la donnée de f à cette position sera celle de la ..... écriture.

— Si P_1 et P_2 lisent dans f à la même position, les données lues seront les mêmes si il n'y a pas eu ..... dans f à cette position entre les .....

— Si P_1 fait une lecture et P_2 une écriture dans f à la position ℓ, P_1 reçoit la donnée écrite par P_2 si :

i. la lecture de P_1 est faite ..... l'écriture de P_2,

ii. il n'y a pas eu d'autre ..... entre la lecture de P_1 et l'écriture de P_2.

e. Terminez les 4 processus.

4. Expérimentation de `mycat` sur des flux FIFO.

a. Créez un fichier `fifo`, correspondant à un pipe nommé.

```
sh$ mkfifo fifo
```

b. Dans quatre terminaux différents, lancez deux écrivains et deux lecteurs sur le fichier `fifo`.

```
sh$ xterm -e bash -c "./mycat > fifo" &
sh$ xterm -e bash -c "./mycat > fifo" &
sh$ xterm -e bash -c "./mycat < fifo" &
sh$ xterm -e bash -c "./mycat < fifo" &
```

c. Tapez les entrées suivantes dans les fenêtres des écrivains, en regardant leurs résultats dans les fenêtres des lecteurs.

aaaa<ENTER>bbbb<ENTER>	dans la fenêtre du premier écrivain.
cccc<ENTER>dddd<ENTER>	dans la fenêtre du deuxième écrivain.
eeee<ENTER>ffff<ENTER>	dans la fenêtre du premier écrivain.

d. Tuez les processus écrivains.

e. Relancez un nouveau écrivain et tapez gggg<ENTER> dans son terminal.

f. Tuez les processus lecteurs.

g. Relancez un nouveau écrivain et tapez `hhhh`<ENTER> dans son terminal.

h. Soit P_1 et P_2 deux processus et f un fichier FIFO :

— Une lecture sur f est bloquante, si f est ..... et qu'il existe encore un ..... .

— Une écriture sur f est bloquante, si f est pleine et qu'il existe encore un ..... .

— En absence de lecteur, une écriture sur une f génère l'émission du signal ..... . Si ce signal n'est ni attrapé, ni ignoré, ceci entraine ..... du processus. Si ce signal est attrapé ou ignoré, l'écriture renvoie une ..... et la variable ..... reçoit la valeur `EPIPE`.

— Si P_1 et P_2 écrivent dans f, les données seront écrites soit l'une derrière l'autre, soit enchevêtrées.

— Si P_1 et P_2 lisent dans f, ils ne peuvent pas lire la ..... donnée.

— Si P_1 fait une lecture et P_2 une écriture dans f, P_1 reçoit la donnée écrite par P_2 si :

i. la FIFO f est ..... ,

ii. il n'y a pas d'autre ..... entre la lecture de P_1 et l'écriture de P_2,

iii. il n'y a pas d'autre ..... entre la lecture de P_1 et l'écriture de P_2.

Si ces conditions sont réalisées, l'ordre chronologique de la lecture de P_1 et l'écriture de $P2$ ..... .

Exercice 2 - `fork`, `wait`, `pipe`

Le but de cet exercice est de reprendre ce qui a été fait dans l'exercice 5 du TP 4, et de proposer une nouvelle variante reposant cette fois sur des *pipes*.

1. Ecrivez la variante `wc-par-v3.c`, dans laquelle le père ouvre un *pipe* avant de créer les 4 fils.

   Les fils devront écrire leurs résultats dans ce *pipe*, de sorte que le père puisse les récupérer et en faire la somme.

2. Si vous avez du temps, écrivez la variante `wc-par.c`, dans laquelle le nombre fils passe de 4 à N.

   Faites varier N sur des exemples pour trouver la valeur optimale.

Exercice 3 - `pipe`, `fcntl`

L'appel système

```
int pipe(int fd[2])
```

permet de créer un fichier FIFO (aussi appelé *pipe*) non nommé.

En cas de succès, le descripteur `fd[0]` permet de lire le contenu du pipe, et le descripteur `fd[1]` permet d'écrire dans le pipe.

Le but de cet exercice est de déterminer la taille (en octets) d'un pipe non nommé.

1. Une approche naïve mais facile à mettre en place est d'écrire un code C qui :
   a. crée un pipe non nommé ;
   b. boucle indéfiniment, en écrivant à chaque étape un caractère dans le pipe ;
   c. affiche à l'écran, avant chaque écriture dans le pipe, le nombre de caractères précédemment écrits dans le pipe.

   Il suffit alors de compiler puis lancer le programme, et d'attendre que l'écriture devienne bloquante (ce qui arrivera lorsque le pipe sera plein).

   Implantez et testez cette approche.

2. Consultez le man de la fonction `fcntl` pour trouver une meilleure approche. Testez votre nouvelle approche et comparez avec le résultat obtenu à la question précédente.

3. Si vous avez le temps, adaptez votre code pour déterminer la taille (en octets) d'un pipe nommé.

   note : Vous devrez faire appel aux fonctions `mkfifo` et `unlink`.

Exercice 4 - Analyse de l'exécution d'une commande shell (examen 2023-2024)

On considère dans cet exercice la commande shell suivante [1] :

```
sh$ cat | while read line ; do
            if [ "$line" = "q" ] ; then break ; else echo "got: '$line'" ; fi
          done
Bonjour
got: 'Bonjour'
Au revoir
got: 'Au revoir'
q
q
sh$
```

On peut suivre ici un exemple d'exécution de cette commande, sachant que les textes surlignés sont les données saisies au clavier par l'utilisateur, et que les textes non surlignés sont les sorties produites par la commande.

On constate que l'exécution se passe à peu près comme prévu : l'utilisateur saisit des lignes de texte qui sont ensuite réécrites à l'écran précédées de `"got: "`, et ce jusqu'à la saisie d'une ligne contenant uniquement la lettre `q`. Cependant, la commande ne s'arrête pas tout à fait comme prévu. En effet, l'arrêt ne survient qu'après la **deuxième** saisie de la lettre `q`.

Afin d'expliquer ce phénomène, nous allons dans un premier temps écrire un programme `C` équivalent à la commande shell. Il sera alors possible d'analyser l'exécution de ce programme `C` en utilisant ce qui a été vu en cours concernant les flux et les *pipes*. Pour vous aider, une ébauche du programme est fourni à la figure 1 et dans le fichier `/pub/FISE_OSSE11/exam/cat_bug_template.c` .

1. Complétez le code de la fonction `left_child`, correspondant à l'appel à `cat`.

   note : Comme indiqué dans le commentaire, on récupérera et on traitera les caractères un par un. De plus, on utilisera des **flux noyau** pour cette question.

2. Complétez le code de la fonction `right_child`, correspondant à la partie à droite du symbole `|` de la commande.

3. Complétez le code de la fonction `main`.

4. On suppose pour cette question que le programme `C` a été compilé, puis exécuté, et que l'utilisateur vient de saisir une ligne contenant uniquement la lettre `q`.

   Combien a-t-on de processus en cours d'exécution avant le traitement de cette ligne ? et après ?

5. On suppose pour cette question que le programme `C` a été compilé, puis exécuté, et que l'utilisateur vient de saisir deux lignes consécutives contenant uniquement la lettre `q`.

   Analysez ce qui se passe au niveau du *pipe* correspondant à la variable `pipefd` et expliquez pourquoi le programme principal (`main`) s'arrête.

6. Que se passe-t-il si on oublie de fermer le flux d'écriture de `pipefd` dans le code de `right_child` ?

7. Que se passe-t-il si on oublie de fermer les flux de `pipefd` dans le code de `main` ?

1. Cette commande est une version simplifiée d'une commande rencontrée en production. Dans la version originale, il y a à droite du symbole `|` un appel à `gdb` (un déboguer pour le langage C, qui lit des instructions sur l'entrée standard et s'arrête à la réception du mot-clé `quit`). La partie à gauche contient quant à elle plusieurs appels à `echo` en plus de `cat`, permettant d'envoyer des instructions à `gdb` en plus de celles saisies par l'utilisateur. Le problème rencontré est alors le même que celui de cet exercice, à savoir que la commande ne s'arrête pas si on entre seulement une fois le mot-clé `quit`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int pipefd[2];

void left_child() {
    /* close the read end of pipefd, since this child will not use it */
    ...
    /* while we can read a character on the standard input,
     * get it and write it on the write end of pipefd */
    ...

    exit(0);
}

void right_child() {
    /* close the write end of pipefd, since this child will not use it */
    ...

    char line[1024];
    int index = 0;
    while (1) {
        /* get a line from the read end of pipefd,
         * or abort if the maximum size of 1024 is reached */
        index = 0;
        do {
            /* read a character from the read end of pipefd */
            ...

            index++;
            if (index > 1024) {
                fprintf(stderr, "error: input is too long\n");
                exit(1);
            }
        } while (line[index-1] != '\n');

        /* check whether we should stop reading or print the line */
        if (index == 2 && line[0] == 'q') {
            break;
        } else {
            /* write the expected output on the standard output */
            ...
        }
    }
    exit(0);
}

int main() {
    /* initialize pipefd */
    ...
    /* create/run the left and right children */
    ...
    /* close both ends of pipefd, since the main program will not use them */
    ...
    /* wait the termination of the two children */
    ...

    return 0;
}
```

FIGURE 1 – Ébauche du programme `C` demandé à l'exercice 4

Exercice 5 - Mini shell

L'objectif de cet exercice est la réalisation d'un mini shell.

On va se concentrer ici sur les problématiques de programmation système soulevées par la conception d'un shell. En revanche, on ne traitera pas de la gestion précise de la syntaxe shell (de fait, pour faire les choses correctement, il faudrait utiliser les outils présentés dans l'UE `ASCO23` au semestre 3).

En fait, on va même s'autoriser à utiliser une syntaxe un peu différente et très simplifiée :

- On ne lancera que des commandes sans arguments ;
- On ne gérera pas les variables, sauf quelques cas très particuliers ;
- Les symboles `<`, `>`, `|` et `&` seront placés au début des commandes, de sorte que le traitement à effectuer puisse être déterminé en fonction du premier caractère saisi par l'utilisateur ;
- Les redirections seront mises en place en amont des commandes, sur des lignes dédiées.

1. Écrivez le code `minishell.c` d'un programme qui lit des commandes sur le flux standard d'entrée et les exécute en avant plan :
   - Le programme attend la fin de la commande courante avant de lire et exécuter la suivante ;
   - Les commandes n'ont pas d'arguments ;
   - Le programme écrit un message d'erreur explicite si la commande saisie n'a pas été trouvée ;
   - Le programme s'arrête sur la fin du fichier standard d'entrée ou si la commande **exit** est saisie.

   On pourra faire des tests avec des commandes comme `ls`, `cat`, `yes` ou encore `./a.out` après avoir compilé un fichier source en `C` de votre choix.

2. Complétez le programme précédent pour gérer les commandes suivantes :

   `$$` qui affiche le PID du programme,
   `$?` qui affiche le code de retour de la dernière commande exécutée.

3. Complétez le programme pour gérer les *redirections* :

   `>file` redirigera le flux standard de sortie de la prochaine commande dans `file`,
   `<file` redirigera le flux standard d'entrée de la prochaine commande afin de lire dans `file`.

   On veillera à ce que la redirection soit appliquée **uniquement** à la prochaine commande. Et on ne gérera que le cas où il n'y a pas d'espace entre le symbole de redirection et le nom du fichier.

4. Complétez le programme pour introduire la *mise en arrière-plan* :

   `&cmd` lancera la commande `cmd` en arrière plan.

   Le programme lira alors la commande suivante sans attendre.

5. Complétez le programme pour que la commande `fg` fasse passer en avant plan la dernière commande lancée en arrière plan.

   note : Dans un premier temps, on pourra se contenter de gérer uniquement le cas d'un unique processus en arrière plan, et renvoyer un message d'erreur si l'utilisateur demande de lancer une seconde commande en arrière plan.

6. Complétez le programme pour que la suite de commandes `|cmd1` puis `cmd2` lance `cmd1` et `cmd2`, avec le flux de sortie standard de `cmd1` utilisé comme flux d'entrée standard pour `cmd2`.

   Il s'agit donc d'avoir un comportement équivalent à

   sh$ cmd1 | cmd2

   dans un shell standard.

   On pourra tester avec `|top`, suivi de `head` / `tail` / `less`.

7. Complétez le programme pour que la saisie de <CTRL-C>, qui génère le signal `SIGINT`, ne termine ni `minishell`, ni les processus en arrière plan, mais uniquement le processus en avant plan si il existe.

> **aide :** Pour cela, il faut que
>
> 1. le processus `minishell` ignore le signal `SIGINT`,
> 2. le processus en avant plan traite le signal `SIGINT` avec le comportement par défaut,
> 3. le signal `SIGINT` ne soit pas transmis aux processus en arrière plan.
>
> Une manière de réaliser le dernier point est de faire appel à `setpgid(0, 0);` ce qui a pour effet de placer le processus courant dans un nouveau groupe de processus.

8. Complétez le programme précédent pour que la saisie de <CTRL-Z> suspende le processus en avant plan uniquement, et pas le `minishell` ni les processus en arrière plan.

> **aide :**
>
> 1. Le noyau envoie le signal `SIGSTP` à tous les processus du groupe de processus ciblé.
> 2. Le processus `minishell` doit attraper ce signal.
> 3. Le processus en avant plan doit avoir le traitement par défaut (suspendre) pour ce signal et appartenir au groupe qui contrôle le tty. Un appel à `fork` met automatiquement le fils dans le groupe du père.
> 4. Les processus en arrière plan ne doivent plus appartenir au groupe (ceci a normalement déjà été mis en place à la question précédente). Ils ne reçoivent alors pas le signal `SIGTSTP`.
> 5. La commande `fg` fait repasser en avant plan soit le processus en avant plan précédemment suspendu, soit le dernier processus lancé en arrière plan. Dans le deuxième cas, il faut remettre le processus dans le groupe pour que <CTRL-Z> soit de nouveau actif. Pour ce faire, on utilisera les fonctions `getpgid` et `tcsetpgrp`.