MINITALK

"The purpose of this project is to code a small data exchange program using UNIX signals."

I. Les règles :

Les règles à respecter pour ce projet sont les suivants :

- Le projet doit **respecter la norme** donnée par 42.
- **Les fonctions ne doivent pas s'interrompre soudainement** (segmentation fault, bus error, double free, etc) hormis pour certains comportements indéterminés : **les erreurs doivent être gérer.**
- **Tout espace alloué dynamiquement doit être libéré si nécessaire**. Par conséquent, aucun leaks ne doit être présent dans le projet final.
- Si le projet requiert une **Makefile**, celui-ci doit compiler mes sources avec les **flags -Wall, -Wextra et -Werror** (le Makefile ne doit pas relinker). Il doit aussi contenir les règles suivantes : **$(NAME), all, clean, fclean, re et bonus** si ces derniers sont réalisés.
- **Les bonus doivent se trouver dans un fichier indépendant.**
- Si le projet le permet, **les sources de la Libft doivent être copiées et associées au Makefile dans un fichier libft.** La libft **doit être compilée grâce au Makefile.**
- Les **exécutables** doivent se nommer : **"client" et "server."**
- Les **fonctions autorisées** sont les suivantes : write, signal, sigemptyset, sigaddset, sigaction, kill, getpid, malloc, free, pause, sleep, usleep, exit.

II. Fonctions Autorisées :

<table>
<tr><th colspan="2">WRITE</th></tr>
<tr><td colspan="2">http://manpagesfr.free.fr/man/man2/write.2.html</td></tr>
<tr><td>unistd.h</td><td>ssize_t write(int fd, const void *buf, size_t count);</td></tr>
<tr><td colspan="2">write() écrit jusqu'à count octets dans le fichier associé au descripteur fd depuis le tampon pointé par buf.</td></tr>
<tr><th colspan="2">SIGNAL</th></tr>
<tr><td colspan="2">http://manpagesfr.free.fr/man/man2/signal.2.html</td></tr>
<tr><td>signal.h</td><td>typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);</td></tr>
<tr><td colspan="2">signal() installe le gestionnaire handler pour le signal signum.

handler peut être SIG_IGN, SIG_DFL ou l'adresse d'une fonction définie par le programmeur (un « gestionnaire de signal »).

Lors de l'arrivée d'un signal correspondant au numéro signum, un des événements suivants se produit :
- Si le gestionnaire est SIG_IGN, le signal est ignoré.
- Si le gestionnaire est SIG_DFL, l'action par défaut associé à ce signal est entreprise : Term, terminer le processus ; Ign, ignore le signal ; Core, termine le processus et créer un fichier core ; Stop, terminer le processus ; et Cont, continuer le processus s’il est arrêté.
- Si le gestionnaire est une fonction, alors tout d'abord le gestionnaire est reconfiguré à SIG_DFL, ou le signal est bloqué, puis handler est appelée avec l'argument signum.

A noter que, si l'invocation du gestionnaire fait que le signal est bloqué, le signal est débloqué au retour du gestionnaire.

Attention, les signaux SIGKILL et SIGSTOP ne peuvent être ni ignorés, ni interceptés.</td></tr>
<tr><th colspan="2">SIGEMPTYSET & SIGADDSET</th></tr>
<tr><td colspan="2">http://manpagesfr.free.fr/man/man3/sigsetops.3.html</td></tr>
<tr><td>signal.h</td><td>int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);</td></tr>
<tr><td colspan="2">Ces fonctions permettent de manipuler un ensemble de signaux POSIX.

Plus précisément :
- sigemptyset() initialise l’ensemble des signaux donnés par set à vide, avec tous les signaux exclus de l’ensemble.
- sigaddset() ajoutent le signe du signal de l’ensemble.</td></tr>
</table>

SiGACTION	
http://manpagesfr.free.fr/man/man2/sigaction.2.html	
signal.h	int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

sigaction() sert à modifier l'action effectuée par un processus à la réception d'un signal spécifique. Signum indique le signal concerné, à l'exception de SIGKILL et SIGSTOP.

KILL	
http://manpagesfr.free.fr/man/man2/kill.2.html	
sys/types.h **signal.h**	int kill(pid_t pid, int sig);

kill() peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus :

- Si **pid est positi**f, le signal sig est envoyé au processus dont l'identifiant est indiqué par pid.
- Si **pid vaut zéro**, alors le signal sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.
- Si **pid vaut -1**, alors le signal sig est envoyé à tous les processus pour lesquels le processus appelant à la permission d'envoyer des signaux, sauf celui de PID 1 (init), mais voir plus bas.
- Si **pid est inférieur à -1**, alors le signal sig est envoyé à tous les processus du groupe dont l'identifiant est -pid.
- Si **sig vaut 0,** aucun signal n'est envoyé mais les conditions d'erreur sont vérifiées ; cela peut être utilisé pour vérifier l'existence d'un identifiant de processus ou de groupe de processus.

GETPID	
http://manpagesfr.free.fr/man/man2/getpid.2.html	
sys/types.h **signal.h**	pid_t getpid(void);

getpid() renvoie l'identifiant du processus appelant. Ceci est souvent utilisé par des routines qui créent des fichiers temporaires uniques.

MALLOC & FREE	
http://manpagesfr.free.fr/man/man3/malloc.3.html http://manpagesfr.free.fr/man/man1/free.1.html	
stdlib.h	void *malloc(size_t *size*); void free(void *ptr);

Respectivement :

- **malloc()** alloue size octets, et renvoie un pointeur sur la mémoire allouée.
- **free()** libère l'espace mémoire pointé par ptr, qui a été obtenu lors d'un appel antérieur à malloc().

PAUSE	
http://manpagesfr.free.fr/man/man2/pause.2.html	
unistd.h	int pause(void);
pause() force le processus (ou le thread) appelant à s'endormir jusqu'à ce qu'un signal soit reçu, qui le termine ou lui fasse invoquer une fonction de gestionnaire de signal.	
SLEEP & USLEEP	
http://manpagesfr.free.fr/man/man3/sleep.3.html http://manpagesfr.free.fr/man/man3/usleep.3.html	
unistd.h	unsigned int sleep(unsigned int nb_sec); int usleep(useconds_t usec);
Respectivement : - **sleep()** endort le processus appelant jusqu'à ce que nb_sec secondes se soient écoulées, ou jusqu'à ce qu'un signal non ignoré soit reçu. - **usleep()** suspend l'exécution du programme appelant durant (au moins) usec microsecondes. La période de sommeil peut être allongée par la charge système, par le temps passé à traiter l'appel de fonction, ou par la granularité des temporisations système.	
EXIT	
http://manpagesfr.free.fr/man/man3/exit.3.html	
stdlib.h	void exit(int status);
exit() termine normalement le processus et la valeur status & 0377 est renvoyée au processus parent.	

III. Partie Obligatoire :

Le projet consiste à **créer un programme de communication sous la forme d'un client et d'un serveur** :

- Le **serveur doit être lancé en premier**, et après avoir été lancé, il doit **afficher son PID** ;
- Le **client prend comme paramètres : le PID du serveur** en question et **la chaîne de caractère qui doit être envoyée**. Le client doit, par la suite, communiquer **la chaîne passé en paramètre au serveur**. Une fois **la chaîne reçue, le serveur doit l'afficher.**

La communication doit être effectuée uniquement à l'aide des signaux UNIX.

Le serveur réalisé durant ce projet **doit pouvoir recevoir des chaînes de plusieurs clients à la suite,** sans besoin d'être redémarré.

Attention, le serveur doit pouvoir afficher la chaîne **assez rapidement.** Pour référence, une 1 seconde pour 100 caractères est un temps colossal. De plus, les seuls signaux autorisés sont **SIGUSR1 et SUGUSR2.**

Ce qu'envoie le client s'appelle un paquet. Un paquet est composé de :

- **IP Source ;**
- **IP Destination ;**
- **MAC Source ;**
- **MAC Destination ;**
- **Le protocole utilisée ;**
- **Le message ;**

IV. Signaux Unix :

Sur un système Unix, il est fréquent que des **processus différents doivent communiquer entre eux et coordonner leurs activités**. Historiquement, le **1er mécanisme de coordination entre processus à été l'utilisation des signaux**. A noter que, sans le savoir, nous avons déjà utilisé certains signaux sans les avoir réellement détaillés explicitement : par exemple, la commande kill(1).

Lorsque 2 processus ont un ancêtre commun, il est possible de les relier en utilisant un pipe. Dans le cas contraire, il est possible d'utiliser un **fifo pour obtenir un effet similaire** :

- Un pipe crée un tube, un canal, unidirectionnel de données qui peut être utilisé pour la communication entre processus. Le tableau pipefd est utilisé pour renvoyer deux descripteurs de fichier faisant référence aux extrémités du tube (pipefd[0] fait référence à l'extrémité de lecteur du tube, pipefd[1] en revanche fait référence à l'extrémité d'écriture du tube). Les données écrites sur l'extrémité d'écriture du tubes sont mises en mémoire tampon par le noyau jusqu'à ce qu'elles soient lues sur l'extrémité de lecture du tubes.
- Un fifo peut être créée en utilisant la commande mkfifo(1), mkfifo(2) ou bien encore mkfifo(3) qui s'utilisent tous les 3 comme l'appel système open(2), il faudra ensuite lui donner un nom sous la forme d'une chaîne de caractères et tout processus qui connait le nom de la fifo pourra ensuite l'utiliser.

Les **sémaphores** que nous avons utilisés pour **coordonner plusieurs threads** sont **également utilisables entre processus moyennant quelques différences.** A savoir qu'une sémaphores est une variable (ou un type de donnée abstrait) partagée par différents « acteurs », qui garantit que ceux-ci ne peuvent y accéder que de façon séquentielle à travers des opérations atomiques, et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) et synchroniser les processus dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra 1 et utilisé pour la première fois dans le système d'exploitation THE Operating system.

Enfin, **des processus liés ou non doivent parfois accéder aux mêmes fichiers.** Ces accès concurrents, s' ils ne sont pas correctement coordonnés, peuvent conduire à des corruptions de fichiers.

L'envoi et la réception de signaux est le mécanisme de communication entre processus le plus primitif sous Unix. Un signal est une forme d'interruption logicielle. Un **microprocesseur va justement utiliser ces interruptions pour permettre au système d'exploitation de réagir aux événements extérieurs qui surviennent, et aux demandes d'action de la part du processeur ou d'un programme.**

Plus précisément, u**n signal Unix est un mécanisme qui permet à un processus de réagir de façon asynchrone à un événement qui s'est produit.** Certains de ces événements sont directement liés au fonctionnement du matériel. D'autres sont provoqués par le processus lui-même ou un autre processus s'exécutant sur le système. Attention, il faut savoir qu'il existe 2 types de signaux :

- **Un signal synchrone est un signal qui a été directement causé par l'exécution d'une instruction du processus.** Un exemple typique de signal synchrone est le signal SIGFPE qui est généré par le système d'exploitation lorsqu'un processus provoque une exception lors du calcul d'expressions mathématiques. C'est le cas notamment lors d'une division par zéro. La sortie ci-dessous illustre ce qu'il se produit lors de l'exécution du programme /Fichiers/src/sigfpe.c.
- **Un signal asynchrone est un signal qui n'a pas été directement causé par l'exécution d'une instruction du processus.** Il peut être produit par le système d'exploitation ou généré par un autre processus comme lorsque nous avons utilisé kill(1) dans un shell pour terminer un processus.

La liste des signaux est définie et certains signaux sont bien connus. Le signal 9 (kill) est sans doute le plus célèbre car il indique au processus qu'il doit se terminer immédiatement. Enfin, chaque processus peut définir le comportement à adopter lors de la réception de chaque type de signal. Voici la liste des des signaux disponibles sous Unix/Linux:

Num	Nom	Description
1	SIGHUP	Instruction (HANG UP) - Fin de session
2	SIGINT	Interruption
3	SIGQUIT	Instruction (QUIT)
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace trap
6	SIGABRT (ANSI)	Instruction (ABORT)
6	SIGIOT (BSD)	IOT Trap
7	SIGBUS	Bus error
8	SIGFPE	Floating-point exception - Exception arithmétique
9	SIGKILL	Instruction (KILL) - termine le processus immédiatement
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Violation de mémoire
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Broken PIPE - Erreur PIPE sans lecteur
14	SIGALRM	Alarme horloge
15	SIGTERM	Signal de terminaison
16	SIGSTKFLT	Stack Fault
17	SIGCHLD ou SIGCLD	modification du statut d'un processus fils
18	SIGCONT	Demande de reprise du processus
19	SIGSTOP	Demande de suspension imbloquable
20	SIGTSTP	Demande de suspension depuis le clavier

Num	Nom	Description
21	SIGTTIN	Lecture terminal en arrière-plan
22	SIGTTOU	Ecriture terminal en arrière-plan
23	SIGURG	Événement urgent sur socket
24	SIGXCPU	Temps maximum CPU écoulé
25	SIGXFSZ	Taille maximale de fichier atteinte
26	SIGVTALRM	Alarme horloge virtuelle
27	SIGPROF	Profiling alarm clock
28	SIGWINCH	Changement de taille de fenêtre
29	SIGPOLL (System V)	Occurrence d'un événement attendu
29	SIGIO (BSD)	I/O possible actuellement
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système
31	SIGUNUSED	