CPE Lyon - 3ICS - Année 2021/22

Administration des systèmes Linux

TD5 – Communication inter-processus

Ce TD aborde les différentes approches permettant a deux processus de communiquer et d'échanger des données.

METRAL Emile ICS3

1. Signaux

Les signaux sont des mécanismes permettant de manipuler et de communiquer avec des processus sous Linux. Le sujet des signaux est vaste ; nous traiterons ici quelques uns des signaux et techniques utilisés pour contrôler les processus.

Un signal est un message spécial envoyé à un processus. Les signaux sont asynchrones lorsqu'un processus reçoit un signal, il le traite immédiatement, sans même terminer la fonction ou la ligne de code en cours .

Il y a plusieurs douzaines de signaux différents, chacun ayant une signification différente. Chaque type de signal est caractérisé par son numéro de signal, mais au sein des programmes, on y fait souvent référence par un nom. Sous Linux, ils sont définis dans /usr/include/bits/signum.h (vous ne devriez pas inclure ce fichier directement dans vos programmes, utilisez plutôt <signal.h>).

Lorsqu'un processus reçoit un signal, il peut agir de différentes façons, selon l'action enregistrée pour le signal. Pour chaque signal, il existe une action par défaut, qui détermine ce qui arrive au processus si le programme ne spécifie pas d'autre comportement. Pour la plupart des signaux, le programme peut indiquer un autre comportement – soit ignorer le signal, soit appeler un gestionnaire de signal, fonction chargée de traiter le signal. Si un gestionnaire de signal est utilisé, le programme en cours d'exécution est suspendu, le gestionnaire est exécuté, puis, une fois celui-ci terminé, le programme reprend.

Le système Linux envoie des signaux aux processus en réponse à des conditions spécifiques.

Par exemple, SIGBUS (erreur de bus), SIGSEGV (erreur de segmentation) et SIGFPE (exception de virgule flottante) peuvent être envoyés à un programme essayant d'effectuer une action non autorisée. L'action par défaut pour ces trois signaux est de terminer le processus et de produire un ficher core. (coredump)

Comme les signaux sont asynchrones, le programme principal peut être dans un état très fragile lorsque le signal est traité et donc pendant l'exécution du gestionnaire de signal. C'est pourquoi vous devriez éviter d'effectuer des opérations d'entrées/sorties ou d'appeler la plupart des fonctions système ou de la

bibliothèque C depuis un gestionnaire de signal. Un gestionnaire de signal doit effectuer le minimum nécessaire au traitement du signal, puis repasser le contrôle au programme principal (ou terminer le programme). Dans la plupart des cas, cela consiste simplement à enregistrer que le signal est survenu. Le programme principal vérifie alors périodiquement si un signal a été reçu et réagit en conséquence.

Il est possible qu'un gestionnaire de signal soit interrompu par l'arrivée d'un autre signal.

Bien que cela puisse sembler être un cas rare, si cela arrive, il peut être très difficile de diagnostiquer et résoudre le problème Un processus peut également envoyer des signaux à un autre processus. Une utilisation courante de ce mécanisme est de terminer un autre processus en lui envoyant un signal SIGTERM ou SIGKILL . Une autre utilisation courante est d'envoyer une commande à un programme en cours d'exécution. Deux signaux "définis par l'utilisateur" sont réservés à cet effet : SIGUSR1 et SIGUSR2. Le signal SIGHUP est également parfois utilisé dans ce but, habituellement pour réveiller un programme inactif ou provoquer une relecture du fichier de configuration.

- [1A] La fonction sigaction permet de définir un handler pour un signal. En respectant les points particuliers mis en exergue dans les paragraphes précédents vous devez :

Écrire un programme qui se "forke" en 5 enfants.

Toutes les secondes sur une durée de 30 secondes, le processus père doit envoyer un signal SIGUSR1 a l'un des enfants.

Le processus recevant le signal SIGUSR1 doit afficher le message « Hello from PID », PID étant le pid du processus enfant.

Une fois les 30 secondes écoule, le processus parent doit envoyer un signal SIGINT aux 5 processus enfants (Attention aux processus Zombie)

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>

int childProcess()
{
    while (1)
    {
        sleep(1);
    }
    // exit(0);
}

void handler(int signum)
{
    printf(" Hello from PID %d\n", (int)getpid());
}

int main(int argc, char *argv[])
```

```
{

    /* fork a child process */
    struct sigaction action;
    pid_t pid[5] = {0};
    int i = 0;
    int nbrFils = 5;
    action.sa_handler = handler;

    sigaction(SIGUSR1, &action, NULL);

    for (i = 0; i < nbrFils; i++)
    {
        pid[i] = fork();
        srand(time(0));
        if (pid[i] < 0)
        {
            fprintf(stderr, "Fork Failed");
            return 1;
        }

        else if (pid[i] == 0)
        {
            childProcess();
        }

    }
    for (int n = 0; n < 30; n++)
    {
        int num = rand() % 5;
        kill(pid[num], SIGUSR1);
        sleep(1);
    }
    for (int p = 0; p < 5; p++)
    {
        kill(pid[p], SIGINT);
    }

}
```

- [1B] Il existe une fonction signal qui permet de faire presque la même chose que sigaction. Pourtant son utilisation est fortement déconseillé. Pourquoi ?

```
La fonction signal() n’empêche pas (nécessairement) les autres signaux d’arriver
pendant l’exécution du gestionnaire actuel; sigaction() peut bloquer d’autres
signaux jusqu’à ce que le gestionnaire actuel revienne.

La fonction signal() réinitialise (habituellement) l’action signal à SIG_DFL (par
défaut) pour presque tous les signaux.
```

```
Le comportement exact du signal() varie selon les systèmes - et les normes
permettent ces variations.
```

La communication inter-processus (inter-process communication, IPC) consiste à transférer des données entre les processus. Il existe au moins cinq types de communication inter-processus :

- La mémoire partagée permet aux processus de communiquer simplement en lisant ou écrivant dans un emplacement mémoire prédéfini.
- La mémoire mappée est similaire à la mémoire partagée, excepté qu'elle est associée à un fichier.
- Les tubes permettent une communication séquentielle d'un processus à l'autre.
- Les files FIFO sont similaires aux tubes excepté que des processus sans lien peuvent communiquer car le tube reçoit un nom dans le système de fichiers.
- Les sockets permettent la communication entre des processus sans lien, pouvant se trouver sur des machines distinctes.

2- Mémoire partagée

Une des méthodes de communication inter processus les plus simples est d'utiliser la mémoire partagée. La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient appelé malloc et avaient obtenu des pointeurs vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification. La mémoire partagée est la forme de communication inter processus la plus rapide car tous les processus partagent la même mémoire. L'accès à cette mémoire partagée est aussi rapide que l'accès à la mémoire non partagée du processus et ne nécessite pas d'appel système ni d'entrée dans le noyau. Elle évite également les copies de données inutiles.

Comme le noyau ne coordonne pas les accès à la mémoire partagée, vous devez mettre en place votre propre synchronisation. Par exemple, un processus ne doit pas effectuer de lecture avant que des données aient été écrites et deux processus ne doivent pas écrire au même emplacement en même temps. Une stratégie courante pour éviter ces conditions de concurrence est d'utiliser des sémaphores.

La commande ipcs affichera la liste des segments mémoire en cours d'utilisation. Nous n'aborderons pas la notion de sémaphore, mais il est possible d'utiliser la mémoire partagée pour qu'un processus père et sont fils échangent des données. La synchronisation pourra se faire via les signaux.

- [2A] En vous basant le fichier TD5-shared_memory.c et en réutilisant le code du TD sur les processus (ou celui de l'exercice précédent) vous devez :

Ecrire un programme qui se « forke » (un seul processus fils)

Le processus parent devra afficher un nombre aléatoire compris entre 0 et 100 et l'écrire dans la mémoire partagée.

Après-quoi le processus parent devra envoyer un signal SIGUSR1 au processus fils pour lui signaler que des données sont disponibles dans la mémoire partagée.

Le processus fils devra lire les données disponibles et afficher son pid en même temps que les données lues.

Après l'envoi de 5 valeurs aléatoires au processus fils, le processus parent mettra fin au processus fils, et se terminera.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/random.h>
#include <signal.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <time.h>

struct sigaction action;
int randomNumber;
int segment_id;
char *shared_memory;

void handler(int signum)
{

    printf("----------------------FILS------------------- \n");
    /* Réattache le segment de mémoire partagée à une adresse différente . */
    shared_memory = (char *)shmat(segment_id, (void *)0x5000000, 0);
    printf("FILS:  mémoire partagée réattachée à l ’ adresse % p \n ",
shared_memory);
    /* Affiche la chaîne de la mémoire partagée . */
    printf(" % s pid: %d\n ", shared_memory, getpid());
    /* Détache le segment de mémoire partagée . */
    shmdt(shared_memory);
}

int processChild()
{
    // Attends le signal
    while (1)
    {
        sleep(1);
    }
}

int main()
{
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x400;
    /* Alloue le segment de mémoire partagée . */
    segment_id = shmget(IPC_PRIVATE, shared_segment_size,
                        IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    /* Attache le segment de mémoire partagée . */
    shared_memory = (char *)shmat(segment_id, 0, 0);
    printf("----------------------PARENTS----------------------\n");
    printf(" PARENTS: mémoire partagée attachée à l ’ adresse % p \n ",
shared_memory);
    /* Détermine la taille du segment . */
    shmctl(segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf(" taille du segment : % d \n ", segment_size);
```

```
    // Fais le lien entre l'action et le signal
    action.sa_handler = handler;
    sigaction(SIGUSR1, &action, NULL);

    // Création du fils
    pid_t pid = fork();

    if (pid == 0) // Action à faire dans le fils
    {
        processChild();
    }

    // Initialisation du temps à 0 pour les valeurs aléatoires
    srand(time(0));

    // Boucle de création et écriture des n nombres aléatoires
    for (int i = 0; i < 5; i++)
    {
        printf("----------------------PARENTS-----------------------\n");
        /* Réattache le segment de mémoire partagée à une adresse différente . */
        shared_memory = (char *)shmat(segment_id, (void *)0x5000000, 0);
        // Génération du nombre aléatoire
        randomNumber = random() % 100 + 1;
        /* Écrit une chaîne dans le segment de mémoire partagée . */
        sprintf(shared_memory, "%d", randomNumber);
        /* Détache le segment de mémoire partagée . */
        shmdt(shared_memory);
        //envoie du signale au fils
        kill(getpid(), SIGUSR1);
    }

    // Libère la mémoire
    shmctl(segment_id, IPC_RMID, 0);
    // Arrête le processus du fils
    kill(pid, SIGINT);

    return (0);
}
```

3- Mémoire mappée

La mémoire mappée permet à différents processus de communiquer via un fichier partagé.

Bien que vous puissiez concevoir l'utilisation de mémoire mappée comme étant à celle d'un segment de mémoire partagée avec un nom, vous devez être conscient qu'il existe des différences techniques. La mémoire mappée peut être utilisée pour la communication inter-processus ou comme un moyen pratique d'accéder au contenu d'un fichier.

Vous pouvez vous représenter la mémoire mappée comme l'allocation d'un tampon contenant la totalité d'un fichier, la lecture du fichier dans le tampon, puis (si le tampon est modifié) l'écriture de celui-ci dans le fichier. Linux gère les opérations de lecture et d'écriture à votre place.

- [3A] Pour mettre en correspondance un fichier ordinaire avec la mémoire d'un processus, utilisez l'appel mmap. Cette fonction accepte 6 paramètres. Donnez le rôle de chacun des paramètres avec les valeurs possibles.

```
void * mmap (void *address, size_t length, int protect, int flags, int filedes,
off_t offset)

    Adresse de départ en mémoire virtuelle.
    Taille de la projection.
    Protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE).
    Drapeau (MAP_SHARED, MAP_PRIVATE, MAP_POPULATE...).
    Descripteur de fichier.
    Position dans ce fichier.
```

- [3B] Que signifie la ligne « PROT_READ | PROT_WRITE » dans le fichier reader.c. C'est pour la lecture et l'écriture de la région cartographiée.
- [3C] A quoi sert le drapeau MAP_SHARED ? Ce drapeau est utilisé pour partager le mappage avec tous les autres processus, qui sont mappés à cet objet.
- [3D] Utilisez les fichiers TD5-reader.c et TD5-writer.c pour écrire deux programmes. Le premier programme écrira sous forme binaire le contenu d'un tableau d'entiers de 5 valeurs aléatoires dans la mémoire mappée. Le second programme devra lire ces valeurs depuis la mémoire mappée, et les afficher. Voir Programmes/Ex3/writer.c et reader.c

reader.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

#define FILE_LENGTH 0x100

int reader(char *file) {
    int fd;
    int * file_memory;

    /* Ouvre le fichier */
    fd = open(file, O_RDWR, S_IRUSR | S_IWUSR);
    /* Met en correspondance le fichier et la mémoire */
    file_memory = mmap(0, FILE_LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    /* Lit l'entier , l'affiche  */
    for (int i = 0; i < 5; i++)
    {
        printf(" valeur : %d \n ", file_memory[i]);
    }
```

```
    //sprintf((char *) file_memory, " %d \n ", integer);
    /* Libère la mémoire ( facultatif car le programme se termine ) . */
    munmap(file_memory, FILE_LENGTH);
    return 0;
}

int main(int argc, char const *argv[])
{
    reader("./test.bin");
    return 0;
}
```

writer.c

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Renvoie un nombre aléatoire compris dans l'intervalle [ low , high ].*/
int random_range(unsigned const low, unsigned const high) {
    unsigned const range = high - low + 1;
    return low + (int) (((double) range) * rand() / (RAND_MAX + 1.0));
}

int writer(char *file) {

    int fd;
    int * file_memory;
    int tab[5];
    const size_t n = sizeof tab / sizeof tab[0];

    /* Initialise le générateur de nombres aléatoires */
    srand(time(NULL));

    /* Prépare un fichier suffisamment long pour contenir le nombre . */
    fd = open(file, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek(fd, FILE_LENGTH + 1, SEEK_SET);
    write(fd, " ", 1);
    lseek(fd, 0, SEEK_SET);
    /* Met en correspondance le fichier et la mémoire . */
    file_memory = mmap(0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    /* Ecrit un entier aléatoire dans la zone mise en correspondance . */

    for (int i = 0; i < 5; i++)
    {
```

```
            file_memory[i] = random_range(-100, 100);
        }


        //sprintf((char *) file_memory, " %d %d %d %d %d \n ", random_range(-100,
    100),random_range(-100, 100),random_range(-100, 100),random_range(-100,
    100),random_range(-100, 100));
        /* Libère la mémoire ( facultatif car le programme se termine ) . */
        munmap(file_memory, FILE_LENGTH);
        return EXIT_SUCCESS;
    }

    int main(int argc, char const *argv[])
    {
        writer("./test.bin");
        return 0;
    }
```

- [3E] Que se passe-t-il si le fichier de « mapping » se trouve sur un partage réseau NTFS ou SMB ? Quelles perspectives entrevoyez vous dans l'usage d'une mémoire mappé par rapport a une mémoire partagée ?

4- Tubes (aka « pipes »)

Un tube est un dispositif de communication qui permet une communication à sens unique. Les données écrites sur l'« extrémité d'écriture » du tube sont lues depuis l'« extrémité de lecture».

Les tubes sont des dispositifs séquentiels ; les données sont toujours lues dans l'ordre où elles ont été écrites. Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus père et fils.

Dans un shell, le symbole | crée un tube. Par exemple, cette commande provoque la création par le shell de deux processus fils, l'un pour ls et l'autre pour less :

ls | less

Pour créer un tube, appelez la fonction pipe (man 3 pipe devrait vous être utile). L'appel à pipe stocke le descripteur de fichier en lecture à l'indice zéro et le descripteur de fichier en écriture à l'indice un.

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

Un appel à pipe crée des descripteurs de fichiers qui ne sont valide qu'au sein du processus appelant et de ses fils. Les descripteurs de fichiers d'un processus ne peuvent être transmis à des processus qui ne lui sont pas liés ; cependant, lorsqu'un processus appelle fork , les descripteurs de fichiers sont copiés dans le nouveau processus. Ainsi, les tubes ne peuvent connecter que des processus liés.

- [4A] Vous devez écrire un programme qui se « forke » :

Le processus parent devra envoyer « n » valeurs aléatoires comprises entre 0 et 9 au processus fils a travers un tube. La valeur de « n » est comprise entre 5 et 20.

Le processus fils devra lire toutes les valeurs qui lui sont transmises, calculer leurs somme et quitter.

Le processus parent doit se terminer lorsque le processus fils a fini sa tache.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>

void childProcess(int p[2], char inbuf[1])
{
    int somme = 0;
    while (1)
    {
        for (int i = 0; i < 3; i++)
        {
            /* read pipe */
            read(p[0], inbuf, 1);
            printf("%s\n", inbuf);
            somme = somme + atoi(inbuf);
        }
        printf("Somme : %d\n", somme);
    }
}

/* Renvoie un nombre aléatoire compris dans l'intervalle [ low , high ].*/
int random_range(unsigned const low, unsigned const high)
{
    unsigned const range = high - low + 1;
    return low + (int)(((double)range) * rand() / (RAND_MAX + 1.0));
}

int main()
{
    char *inbuf = malloc(sizeof(int));
    int p[2], valeur, n;
    pid_t pid;

    if (pipe(p) < 0)
        exit(1);

    pid = fork();

    if (pid == 0)
    {
        childProcess(p, inbuf);
```

```
    }

    /* continued */
    /* write pipe */
    // Initialisation du temps à 0 pour les valeurs aléatoires
    srand(time(0));
    n = random_range(5, 20);

    for (int i = 0; i < n; i++)
    {
        sprintf(inbuf, "%d", random_range(0, 9));
        write(p[1], inbuf, 1);
    }

    // Arrête le processus du fils
    wait(NULL);

    return 0;
}
```

- [4B] Quelle est le rôle/intérêt de la commande dup2 ? La fonction dup2 crée une copie du descripteur de fichier donné et lui attribue un nouvel entier. La fonction dup2 prend un ancien descripteur de fichier à cloner comme premier paramètre et le second paramètre est l'entier pour un nouveau descripteur de fichier. Par conséquent, ces deux descripteurs de fichier pointent vers le même fichier et peuvent être utilisés de manière interchangeable.
- [4C] A quoi servent les commandes popen et pclose ? La fonction popen() engendre un processus en créant un pipe, exécutant un fork(), et en invoquant le shell. La fonction pclose() attend que le processus associé se termine et retourne le statut de sortie de la commande comme retourné par wait4().

5- Tubes nommés (FIFO, aka « named pipe »)

Une file premièr entré, premier sorti (first-in, first-out, FIFO) est un tube qui dispose d'un nom dans le système de fichiers. Tout processus peut ouvrir ou fermer la FIFO ; les processus raccordés aux extrémités du tube n'ont pas à avoir de lien de parenté. Les FIFO sont également appelés canaux nommés (named pipe).

Vous pouvez créer une FIFO via la commande `mkfifo` . Indiquez l'emplacement où elle doit être créée sur la ligne de commande. Par exemple, créez une FIFO dans /tmp/fifo en invoquant ces commandes : `mkfifo`

Par exemple sur un premier terminal :

```
mkfifo /tmp/fifo
cat < /tmp/fifo
```

Dans un second terminal : `echo "Hello world" > /tmp/fifo`

Pour créer une FIFO par programmation, utilisez la fonction mkfifo (man 3 mkfifo ?).

L'accès à une FIFO se fait de la même façon que pour un fichier ordinaire. Pour communiquer via une FIFO, un programme doit l'ouvrir en écriture. Il est possible d'utiliser des fonction d'E/S de bas niveau (open , write , read , close , etc.) ou des fonctions d'E/S de la bibliothèque C (fopen, fprintf, fscanf, fclose, etc.).

- [5A] Vous devez écrire un premier programme appelé send_rand qui écrit « n » valeurs aléatoires dans un tube nommé. Ce programme prendra en argument le nom du tube, et l'option -n qui sera suivi du nombre de valeurs aléatoires a envoyer dans le tube. Ce programme devra créer le tube dans le répertoire /tmp si celui-ci n'existe pas déjà.

Exemple de syntaxe : send_rand mypipe -n 30 qui enverra 30 valeurs aléatoires dans le tube nommé « mypipe »

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#define FILE_LENGTH 0x100

/* Renvoie un nombre aléatoire compris dans l'intervalle [ low , high ].*/
int random_range(unsigned const low, unsigned const high) {
    unsigned const range = high - low + 1;
    return low + (int) (((double) range) * rand() / (RAND_MAX + 1.0));
}

int main(int argc, char const *argv[])
{
    /* Initialise le générateur de nombres aléatoires */
    srand(time(NULL));
    FILE *fd;
    char opt;
    bool errflag = false, i_isset = false, n_isset = false;
    int n ;

    // FIFO file path
    char * myfifo = malloc(80*sizeof(char));
    strcpy(myfifo, "/tmp/") ;
    strcat(myfifo, argv[1]);
    printf("Fichier : %s \n", myfifo);

    // Gestion des arguments
    while ( (opt = getopt(argc, argv, "in:")) != -1 )
    {
        switch (opt)
        {
        case 'n':
            if (!i_isset)
```

```
            {
                n = atoi(optarg);
                n_isset = true;
            } else {
                errflag = true;
            }

            break;

        default:
            break;
        }
    }



    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    // Open FIFO for write only
    fd = fopen(myfifo, "w");

    for (size_t i = 0; i < n; i++)
    {
        fprintf(fd, "%d;",random_range(0, 100));
    }

    fclose(fd);
    //printf("%s \n", argv[3]);

    return 0;
}
```

- [5B] Le second programme appelé get_rand lira toutes les valeurs présentes dans le tube, et affichera la moyenne des valeurs avant de quitter.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#define FILE_LENGTH 0x100


int main(int argc, char const *argv[])
{
```

```
    const char separator[2] = ";";
    char *token;
    int *tokenI = malloc(80 * sizeof(int)), i = 0, somme = 0, moyenne = 0;
    int fd;

    // FIFO file path
    char * myfifo = malloc(80*sizeof(char));
    strcpy(myfifo, "/tmp/") ;
    strcat(myfifo, argv[1]);
    printf("Fichier : %s \n", myfifo);


    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80] = {"0"};

    // Open FIFO for Read only
    fd = open(myfifo, O_RDONLY);

    // Read from FIFO
    read(fd, arr1, sizeof(arr1));

    printf(" %s\n", arr1);

    // get the first token
    token = strtok(arr1, separator);

    // Print the read message
    while (token != NULL )
    {
        printf(" %s\n", token);
        somme += atoi(token);
        token = strtok(NULL, separator);
        i++;
    }


    moyenne = somme / i;


    printf("Moyenne: %d\n", moyenne);
    close(fd);


    return 0;
}
```

6- Socket

Un socket est un dispositif de communication bidirectionnel pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus s'exécutant sur d'autres machines. Nous n'aborderons ici que les sockets UNIX (PF_UNIX) , aussi appelés sockets locaux, que vous avez entrevu dans le TD sur MySQL. Pour les sockets réseaux (PF_INET), vous êtes invité a retourner voir le code concernant les projets du module DEV-INF-2.

Les sockets mettant en relation des processus situés sur le même ordinateur peuvent utiliser l'espace de nommage local représenté par les constantes PF_LOCAL et PF_UNIX.

Ils sont appelés sockets locaux ou sockets de domaine UNIX. Leur adresse de socket, un nom de fichier, n'est utilisée que lors de la création de connexions.

Le nom du socket est spécifié dans une struct sockaddr_un . Vous devez positionner le champ sun_family à AF_LOCAL, qui représente un espace de nommage local. Le champ sun_path spécifie le nom de fichier à utiliser et peut faire au plus 108 octets de long. La longueur réelle de la struct sockaddr_un doit être calculée en utilisant la macro SUN_LEN .

Tout nom de fichier peut être utilisé, mais le processus doit avoir des autorisations d'écriture sur le répertoire qui permettent l'ajout de fichiers. Pour se connecter à un socket, un processus doit avoir des droits en lecture sur le fichier. Même si différents ordinateurs peuvent partager le même système de fichier, seuls des processus s'exécutant sur le même ordinateur peuvent communiquer via les sockets de l'espace de nommage local.

Le seul protocole permis pour l'espace de nommage local est 0.

Comme il est stocké dans un système de fichiers, un socket local est affiché comme un fichier. Avec le type s (srwxrwx—x).

Fichier commun pour les 2 programmes suivants :

```
/**
 * @file conf.h
 * @author Emile METRAL
 * @brief define & include for client.c and server.c
 * @version 0.1
 * @date 2021-12-18
 *
 * @copyright Copyright (c) 2021
 *
 */
#if !defined(CONF_H)
#define CONF_H

#define SOCKET_NAME "/tmp/test6.socket"
#define RESULT_FILE "./tmpfile.bin"
#define CLIENT1_FILE "./tmpclient1.bin"
#define BUFFER_SIZE 12
#define FILE_LENGTH 0x100

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <signal.h>
#include <sys/stat.h>
#include <errno.h>

#endif // CONF_H
```

- [6A] En utilisant les sockets UNIX, vous devez écrire un programme client qui enverra une valeur numérique unique a un programme serveur via les sockets UNIX. Ce programme sera exécuté simultanément 2 fois. Les 2 processus contacteront le même serveur et chacun devra envoyer une valeur a un processus serveur via les sockets.

```
/**
 * @file client.c
 * @author METRAL EMILE
 * @brief Client send number and receive a result
 * @version 0.1
 * @date 2021-12-18
 *
 * @copyright Copyright (c) 2021
 *
 */
#include "conf.h"


int main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int ret;
    int data_socket;
    char buffer[BUFFER_SIZE];

    /* Create local socket. */

    data_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (data_socket == -1)
    {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    /*
     * For portability clear the whole structure, since some
     * implementations have additional (nonstandard) fields in
     * the structure.
     */
```

```
    memset(&addr, 0, sizeof(addr));

    /* Connect socket to socket address. */

    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_NAME, sizeof(addr.sun_path) - 1);

    ret = connect(data_socket, (const struct sockaddr *)&addr,
                  sizeof(addr));
    if (ret == -1)
    {
        fprintf(stderr, "The server is down.\n");
        exit(EXIT_FAILURE);
    }

    /* Send argument. */

    ret = write(data_socket, argv[1], strlen(argv[1]) + 1);
    if (ret == -1)
    {
        perror("write");
        return EXIT_FAILURE;
    }

    /* Receive result. */

    ret = read(data_socket, buffer, sizeof(buffer));

    printf("Result = %s\n", buffer);

    /* Close socket. */
    close(data_socket);

    exit(EXIT_SUCCESS);
}
```

- [6B] Le processus serveur attendra d'avoir deux valeurs numériques. Il devra alors calculer la somme des deux valeurs, l'afficher et aussi la retourner aux processus clients, qui a leurs tours, afficheront cette somme.

```
/**
 * @file serveur.c
 * @author METRAL EMILE
 * @brief Server receive from 2 clients numbers and sum them before send the
result to the 2 clients
 * @version 0.1
 * @date 2021-12-18
 *
 * @copyright Copyright (c) 2021
 *
```

```
 */
#include "conf.h"

void childrenProcess();
int writer(char *file, int value);
int reader(char *file);
void handler(int signum);

int result;
int client1_socket;
struct sigaction action;



int main(int argc, char *argv[])
{
    struct sockaddr_un name;
    int down_flag = 0;
    int ret;
    int connection_socket;
    int data_socket;
    char buffer[BUFFER_SIZE];
    int nbrConnections = -1;
    pid_t pid;

    // Fais le lien entre l'action et le signal
    action.sa_handler = handler;
    sigaction(SIGUSR1, &action, NULL);

    /* Create local socket. */

    connection_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (connection_socket == -1)
    {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    /*
     * For portability clear the whole structure, since some
     * implementations have additional (nonstandard) fields in
     * the structure.
     */

    memset(&name, 0, sizeof(name));

    /* Bind socket to socket name. */

    name.sun_family = AF_UNIX;
    strncpy(name.sun_path, SOCKET_NAME, sizeof(name.sun_path) - 1);

    ret = bind(connection_socket, (const struct sockaddr *)&name,
               sizeof(name));
    if (ret == -1)
```

```
    {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    /*
     * Prepare for accepting connections. The backlog size is set
     * to 20. So while one request is being processed other requests
     * can be waiting.
     */

    ret = listen(connection_socket, 2);
    if (ret == -1)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    /* This is the main loop for handling connections. */
    while (1)
    {
        /* Wait for incoming connection. */
        client1_socket = accept(connection_socket, NULL, NULL);

        result = 0;

        pid = fork();

        if (pid < 0)
        {
            perror("No fork");
            exit(EXIT_FAILURE);
        }

        else if (pid == 0)
        {

            /* Wait for next data packet. */
            ret = read(client1_socket, buffer, sizeof(buffer));

            /* Add received summand. */
            result += atoi(buffer);
            printf("Child %d calculating results %d\n", (int) getpid(), result);
            writer(CLIENT1_FILE, result);
        }

        else
        {
            // Close first client connection for the main process
            close(client1_socket);

            // Wait for second client connection
            data_socket = accept(connection_socket, NULL, NULL);
```

```
            /* Wait for next data packet. */
            ret = read(data_socket, buffer, sizeof(buffer));

            /* Add received summand. */
            result += atoi(buffer);
            result += reader(CLIENT1_FILE);
            writer(RESULT_FILE, result);

            // Sending signal to child process
            kill(pid, SIGUSR1);
            sleep(1);

            /* Send result. */
            sprintf(buffer, "%d", result);
            ret = write(data_socket, buffer, sizeof(buffer));

            /* Close socket. */
            close(data_socket);

            break;
        }
    }

    // Arrête le processus du fils
    printf("Parent of child %d killing him\n", pid);
    kill(pid, SIGINT);

    close(connection_socket);

    /* Unlink the socket. */

    unlink(SOCKET_NAME);

    exit(EXIT_SUCCESS);
}

int writer(char *file, int value)
{

    int fd;
    int *file_memory;
    int tab[5];
    const size_t n = sizeof tab / sizeof tab[0];

    /* Prépare un fichier suffisamment long pour contenir le nombre . */
    fd = open(file, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek(fd, FILE_LENGTH + 1, SEEK_SET);
    write(fd, " ", 1);
    lseek(fd, 0, SEEK_SET);
    /* Met en correspondance le fichier et la mémoire . */
    file_memory = mmap(0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    /* Ecrit la valeur dans la zone mise en correspondance . */
    file_memory[0] = value;
```

```
    /* Libère la mémoire ( facultatif car le programme se termine ) . */
    munmap(file_memory, FILE_LENGTH);
    return EXIT_SUCCESS;
}

int reader(char *file)
{
    int fd;
    int *file_memory;
    int value;

    /* Ouvre le fichier */
    fd = open(file, O_RDWR, S_IRUSR | S_IWUSR);
    /* Met en co rr es pon da nce le fichier et la mémoire */
    file_memory = mmap(0, FILE_LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    /* Lit l'entier , l'affiche  */

    value = file_memory[0];

    // sprintf((char *) file_memory, " %d \n ", integer);
    /* Libère la mémoire ( facultatif car le programme se termine ) . */
    munmap(file_memory, FILE_LENGTH);
    return value;
}

void handler(int signum)
{
    char buffer[BUFFER_SIZE];
    result = reader(RESULT_FILE);

    /* Send result. */
    sprintf(buffer, "%d", result);
    write(client1_socket, buffer, sizeof(buffer));

    /* Close socket. */
    close(client1_socket);
}
```

References :

- https://stackoverflow.com/questions/13669474/multiclient-server-using-fork
- http://www.bruno-garcia.net/www/Unix/
- https://www.geeksforgeeks.org/introduction-of-process-synchronization/?ref=lbp