1. write & read

a. Descripteurs I/O (Input/Output) :

Descripteur = unique id pour un fichier
Correspondent aux fichiers ouverts pour une action (Tout est fichier sur linux)

- **STDOUT_FILENO** : sortie standard
- **STDIN_FILENO** : entrée standard
- **STDERR_FILENO** : sortie d'erreur

```
#include <unistd.h>
```

b. La fonction write(...) permet d'écrire sur une sortie.

```
<unistd.h>
char tampon [] = "Hello";
write(stdout, tampon, 6);
```

- Param1 : sortie sur laquelle écrire
- Param 2 : chaîne a écrire
- Param 3 : nombre d'octet a écrire = nb char (Ne pas oublier \0)

c. La fonction read(...) permet de lire une entrée

```
<unistd.h>
write(STDOUT_FILENO, "Ecrire : ", 10);
char tampon[100];
int nb = read(STDIN_FILENO, tampon, 100);
write(STDOUT_FILENO, "J'ai lu : ", 10);
write(STDOUT_FILENO, tampon, nb);
```

- Param 1 : entrée sur laquelle lire
- Param 2 : chaîne sur laquelle écrire (Buffer)
- Param 3 : nombre d'octets max à lire sur l'entrée
- Valeur de retour : nombre d'octets lus

d. Différence stdout et STDOUT_FILEN0

Descripteur	Librairie	Fonctions qui l'utilisent
STDOUT_FILEN0	<unistd.h>	read / write / system / ...
stdout	<stdio.h>	fprint / fputs / ....
source		

3) system & chdir

a. La fonction system(...) permet d'exécuter une commande shell

<stdlib.h>

```
system("ls"); // Exécuter un ls
system(getenv("SHELL"));
```

Récupère la variable d'environnement qui donne la path vers le programme d'exécution du Shell (/bin/bash) et exécute ce programme.

Ici on ouvre donc un "sous-shell" depuis notre code

b. La fonction chdir(...) est équivalent à la commande cd, elle change le répoertoire courant du processus.

<stdlib.h>

```
chdir("/bin");
system("ls");
```

4) Pointeurs de fonction

```
typedef void (*une_fct)();

void manger() {
   printf("Je mange\n");
}

void dormir(int heures) {
   printf("Je dors %d heures\n", heures);
}

int main() {
   une_fct a = manger;
```

```
    a(); // Appel à manger()
    une_fct b = dormir;
    b(8); // Appel à dormir avec un paramètre
}
```

5) fork & waitpid & getpid

c. La fonction fork() demande au système de dupliquer le processus.

```
pid_t p = fork(); // On duplique le processus ici
if (p == 0) { // p vaut 0 pour le fils
    printf("je suis le processus fils\n");
    exit(EXIT_SUCCESS);
}
printf("je suis le processus père\n");
printf("le processus fils a le numéro %d\n", p);
```

d. La fonction waitpid permet de bloquer l'exécution du processus courant tant qu'un autre processus ne s'est pas terminé

```
<sys/wait.h>
pid_t p = fork();
if (p == 0)
    exit(EXIT_FAILURE);
int status;
waitpid(p, &status, 0);
printf("le s'est terminé avec le status %d\n", status);
```

- Param1 : pid a attendre (-1 pour n'importe quel pid)
- Param2 : adresse vers variable qui stock le status de sortie du PID attendu (Si il s'est terminé avec succès ou avec erreur)
- Param3 : mettre 0, waitpid(3) - Linux man page
- Retour : pid du processus intercepté (Utilise si on met -1 sur param1)

e. La fonction getpid() permet de récupérer le pid du processus courant

6) exec

Les fonctions de la famille **exec** ont pour but de remplacer le processus courant par un autre processus. **Exec** est bloquante => le processus courant attend la fin de l'exécution du sous-processus pour continuer.

<unistd.h>

fonction	Particularité	exemple
execl	path + liste args	execl ("/bin/ls", "ls", "-l", NULL)
execlp	cmd + liste args	execlp("ls", "ls", "-l", NULL)
execv	path + array[] args	execv ("/bin/ls", arg)
execvp	cmd + array[] args	execvp("ls", arg)

avec arg = char * arg[] = { "ls", "-l", NULL};

```
execl ("/bin/ls", "ls", "-la", NULL);
```

- Param1 : path vers le programme à exécuter
- Param x : arguments donnés au programme. La liste d'arguments doit toujours se terminer par NULL
- Return : code de retour du processus qui vient d'être exécuté

7) system() et exec() en détails

- Fait un fork au moment de son exécution
- Le processus enfant fait un **exec** avec la commande donnée dans **system()**
- Il y a en plus un **waitpid** pour attendre la fin de l'exécution de l'enfant.

Du côté de l'**exec** qui est fait dans l'enfant, ce dernier :

- Cherche le fichier exécutable
- Le copie dans son espace mémoire
- Lance son exécution

8) Signaux

Un signal est un message émit à destination d'un processus (ou d'un groupe de processus) pour l'informer que quelque chose vient de survenir ou pour lui demander de réaliser une tâche particulière.
La fonction **signal** permet de définir quelle fonctionne appeler à la réception d'un signal.

Nom	Signal	Description
SIGABRT	Signal Abort	Terminaison anormale, normalement initiée par la fonction abort
SIGFPE	Signal Floating-Point Exception	Opération arithmétique erronée, telle que division du zéro ou opération entraînant un dépassement de capacité. Attention, ce signal peut aussi déclencher suite à des manipulations de valeur numérique sans virgule flottante.
SIGILL	Signal Illegal Instruction	Exécution d'une instruction non conforme. Cela est généralement dû à une corruption du code ou à une tentative d'exécution de données.
SIGINT	Signal Interrupt	Un signal d'interruption généralement généré par l'utilisateur de l'application.
SIGSEGV	Signal Segmentation Violation	Accès à un segment de mémoire virtuelle non mappé en mémoire physique ou tentative de modification d'un segment de mémoire configuré en lecture seule.
SIGTERM	Signal Terminate	Signal de demande de terminaison du programme.
SIGUSR1	User 1	Lié a rien, on peut l'utiliser pour ce que l'on veut

a. La fonction signal

Permet d'inscrire une fonction (void) qui sera appelée à la réception d'un signal :

```
void sig_handler(int sig) {
   printf("recue SIGINT\n");
   exit(0);
}

int main(void) {
   signal(SIGINT, sig_handler); // On enregistre la sig_handler
   while (1) sleep(1); // Attente infinie
}
```

Au moment où le programme recevra le signal **SIGINT** (quand le programme est coupé) il appellera la fonction **sig_handler**

b. Envoyer un signal avec kill

```
kill(pid, SIGUSR1);
```

- param 1 : pid du processus qui doit recevoir le signal

- param 2 : signal à envoyer

9) Redirections

Il est possible de rediriger un fichier vers un autre, par exemple rediriger la sortie standard vers un fichier.

Rediriger STDOUT_FILENO vers un fichier fait que n'importe quel printf sera redirigé vers un fichier plutôt que vers le terminal (Par définition le terminal est aussi un fichier).

<fcntl.h> pour open
<unistd.h> pour dup2

```
// On ouvre le fichier d'entrée
int fdInput = open(argv[1], O_RDONLY);
// On duplique le fichier d'entrée sur l'entrée standard du programme
// (rappel : qui est aussi un fichier)
dup2(fdInput, STDIN_FILENO);
// On ferme le fichier d'entré
close(fdInput);

// On ouvre le fichier de sortie
int fdOut = open(
   argv[2],          // path
   O_CREAT | O_WRONLY, // flags
   0644              // perms (octal)
);
// On duplique le fichier de sortie vers la sortie standards
dup2(fdOut, STDOUT_FILENO);
// On ferme le fichier
close(fdOut);

// On execute la commande tr pour passer de min à maj (Voir exec)
execlp("tr", "tr", "[a-z]", "[A-Z]", NULL);
```

10) Tuyaux

Un pipe permet de faire communiquer deux processus (à travers un tuyaux ou un tube)

Il est composé de deux fichiers dans lesquels ont doit écrire (pour envoyer un message) et lire (recevoir le message).

- Le fichier de sortie = écrire dedans pour envoyer
- Le fichier d'entrée = =lire dedans pour recevoir un message

```
// Création du pipe
int fds[2];
pipe(fds); // Inititalisation

int entree = fds[0];// La case 0 du tableau contient le fichier d'entree
int sortie = fds[1];// La case 1 contient le fichier de sortie



// Création fils (Qui va servir d'emetteur de message)
if (fork() == 0) {
   close(entree); // Ferme l'entrée car le fils n'a pas besoin de lire
   produire(sortie); //Fonction qui permet au fils d'écrire dans le pipe
   exit(EXIT_SUCCESS); // Fin fils
}



// Père (Receveur)
close(sortie); // On ferme le fichier de sortir (le pere ne lis pas)
consommer(entree); // fonction qui permet de lire l'entree du pipe

exit(EXIT_SUCCESS);
```

```
struct Message { // Structure utilisée qui passe à travers le pipe
   int data;
};
```

Fonction pour le fils :

```
void produire(int sortie) { // On passe en param le fichier ou écrire
   for (int i = 0; i < 10; i++) { // On écris 10 fois un message
      struct Message m;
      m.data = i;
      write(sortie, &m, sizeof(m)); // Ecriture dans le pipe
   }
   close(sortie); // On ferme la sortie quand on a terminé
}
```

```
Fonction pour le père

void consommer(int entree) {
   while(1) { // Attente des messages
      struct Message m;
      int r = read(entree, &m, sizeof(m)); // On lis le message
      if (r != sizeof(m)) // Si le message n'est pas correct
         break; // On attend plus (quitte le while)
      printf("%d\n", m.data); // On affiche le contenu du message
   }
   close(entree); // On a terminé, on ferme l'entree
}
```

11) Threads

- Threads : blocks l'exécution en parallèle qui ont la même mémoire
- Equivalent au sous processus avec fork mais moins lourd car mm mémoire
- Sa vie est liée à son père contrairement à avec un fork

```
// Fonction qui va être exécuté dans un ou plusieurs threads en parallèle
void *say_hello(void *data) {
   char *str;
   str = (char*) data; // On cast en string
   while(1) {
      printf("%s\n", str);
      sleep(1);
   }
}

int main() {
   pthread_t t1, t2;
   pthread_create(&t1, NULL, say_hello, "Hello thread 1");
   pthread_create(&t2, NULL, say_hello, "Hello thread 2");
   // Param 1 pointeur vers le thread
   // Param 2 paramètre de setup du thread sous forme
   //       de struct (Souvent NULL = param par défaut)
   // Param 3 fonction à lancer
   //       (Toujours une void, ses param doivent être void*)
   // Param 4 (Argument de la fonction say_hello -> le void* a donner)
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    // t1 et t2 sont liés au processu courant (Celui qui les a lancé)
    // Donc si on kill le programme, tous les thread sont tués
}
```