TP 2 : PROCESSUS

1-1-Gestion des erreurs

Avec la programmation système, nous allons étudier et manipuler tout ce qui touche à votre système d'exploitation. Ainsi, nous devrons faire face assez souvent à des codes d'erreurs. La gestion des erreurs est donc un élément primordial dans la programmation système.

- **La variable globale `errno`**

Pour signaler une erreur, les fonctions renvoient une valeur spéciale, indiquée dans leur documentation. Celle-ci est généralement **-1** (sauf pour quelques exceptions). La valeur d'erreur alerte l'appelant de la survenance d'une erreur, mais elle ne fournit pas la description de ce qui s'est produit. La variable globale **errno** est alors utilisée pour en trouver la cause.

Cette variable est définie dans <errno.h> comme suit :

```
#include <errno.h>

extern int errno;
```

Sa valeur est valable uniquement juste après l'utilisation de la fonction que l'on veut tester. En effet, si on utilise une autre fonction entre le retour que l'on veut tester et l'exploitation de la variable de errno, la valeur de errno peut être modifiée entre temps.

A chaque valeur possible de errno correspond une constante du préprocesseur. Pour les connaître, il faut taper la commande man errno.

- **La fonction perror**

La bibliothèque C met également à notre disposition une fonction permettant d'associer à l'utilisation d'une fonction une description de l'erreur (si il y en a eu une) qui s'est produite. Cette fonction, la voici :

```
#include <stdio.h>

void perror(const char *s);
```

Cette fonction affiche sur stderr (sortie d'erreur standard) une représentation en une chaîne de caractère de l'erreur décrite par errno, précédée par la chaîne de caractère pointée par s ainsi que d'un espace.

Par convention, le nom de la fonction qui a produit cette erreur doit être inclus dans la chaîne. Par exemple :

Exemple :

```
if (fork() == -1) {

    perror("fork");

}
```

1-2-Commandes de gestion des processus

Pour afficher ses propres processus en cours d'exécution, on utilise la commande :

```
$ ps
```

Pour afficher tous les processus en cours d'exécution, on peut utiliser l'option aux (a : processus de tous les utilisateurs ; u : affichage détaillé ; x : démons) :

```
$ ps aux
```

Dans le résultat qui s'affiche, vous pouvez voir la liste de tous vos processus en cours d'exécution.

- La première colonne **USER** correspond à l'utilisateur qui a lancé le processus.
- La deuxième colonne **PID** indique le numéro de PID du processus.
- La huitième colonne **STAT** correspond à l'état du processus.
- La neuvième colonne **START** correspond à l'heure du lancement du processus.
- Enfin, la dernière colonne **COMMAND** correspond au chemin complet de la commande lancée par l'utilisateur (car même si vous lancez un exécutable en mode graphique, vous avez une commande qui s'exécute).

1-3-Création d'un nouveau processus

Bon nombre de fonctions utilisées avec la programmation système (notamment les appels-système) nécessiterons l'inclusion de la bibliothèque **<unistd.h>.** Donc, pensez bien de mettre, au début de vos fichiers :

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

1-3-1-La fonction fork

Pour créer un nouveau processus à partir d'un programme, on utilise la fonction **fork**(qui est un *appel-système*). Pour rappel, le processus d'origine est nommé processus père et le nouveau processus créé processus fils, qui possède un nouveau PID. Les deux ont le **même code source**, mais la valeur retournée par fork nous permet de savoir si l'on est dans le processus père ou dans le processus fils. Ceci permet de faire deux choses différentes dans le processus père et le processus fils (en utilisant une structure de condition ,if ou switch).

La fonction fork retourne une valeur de type **pid_t**. Il s'agit généralement d'un int; il et déclaré dans <sys/types.h>.
Bref, donc... La valeur renvoyée par fork est de :

- **-1** si il y a eu une erreur ;
- **0** si on est dans le processus fils ;
- Le PID du fils si on est dans le processus père. Cela permet ainsi au père de connaître le PID de son fils.

Dans le cas où la fonction a renvoyé-1 et donc qu'il y a eu une erreur, le code de l'erreur est contenue dans la variable globale errno, déclarée dans le fichier errno.h (n'oubliez pas le #include...). Ce code peut correspondre à deux constantes :

- ENOMEM : le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus ;
- EAGAIN : ce code d'erreur peut être dû à deux raisons : soit il n'y a pas suffisamment de ressources systèmes pour créer le processus, soit l'utilisateur a déjà trop de processus en cours d'exécution. Ainsi, que ce soit pour l'une ou pour l'autre raison, vous pouvez rééditer votre demande tant que fork renvoie EAGAIN.

Bon, en résumé, voici le prototype de la fonction fork:

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

1-3-2-Autres fonctions :

- La fonction **getpid** retourne le PID du processus appelant.

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);
```

- La fonction **getppid** retourne le PPID du processus appelant.

```
#include <unistd.h>
#include <sys/types.h>
pid_t getppid(void);
```

- La fonction **getuid** retourne l'UID du processus appelant.

```
   #include <unistd.h>
#include <sys/types.h>

uid_t getuid(void);
```

- La fonction **getgid** retourne le GID du processus appelant.

```
#include <unistd.h>
#include <sys/types.h>
gid_t getgid(void);
```

1-3-3-Code complet :

Bon, maintenant, voici le code complet permettant de créer un nouveau processus et d'afficher des informations le concernant.

```c
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));

    /* On retourne le PID du processus ainsi créé */
    return pid;
}

/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
    printf(" Nous sommes dans le fils !\n"
       " Le PID du fils est %d.\n"
       " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());
}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    child_process();
    break;
    /* Si on est dans le père */
    default:
    father_process(pid);
    break;
    }

    return EXIT_SUCCESS;
}
```

Résultat :

```
$ ./a.out
```

```
Nous sommes dans le père
Le PID du fils est 2972
Le PID du père est 2971
Nous sommes dans le fils
Le PID du fils est 2972
Le PPID du fils est 2971
```

.

1-3-3-Le cast

Ceci vous a peut-être surpris :

```
(int) getpid()
```

Cette technique est appelée **cast** (en français, on traduit par *conversion de type*). Elle permet de convertir une variable d'un type vers un autre type. La syntaxe utilisée est :

```
type nouvelle_variable = (type) ancienne_variable;
```

Dans notre cas ,`getpid` renvoie une variable de type `pid_t`. Or, il n'existe aucune certitude quand au type standard que symbolise `pid_t` (il n'existe pas de formateur spécial pour `scanf` et `printf` dans le but d'afficher une variable de type `pid_t`). Mais comme nous savons que, généralement, `pid_t` prend la forme d'un `int`, on va donc convertir la variable `pid` en type `int`, afin de pouvoir l'afficher grâce à `%d`.

***Attention* : Eviter de placer `fork` dans des boucles infinies. En effet, il n'y a aucune sécurité sur le système : imaginez un peu le désastre si vous vous créiez des milliers de processus...**

1-4-Terminaison d'un processus

Un programme peut se terminer de façon normale (volontaire) ou anormale (erreurs).

1-4-1-Terminaison normale d'un processus

Un programme peut se terminer de deux façons différentes. La plus simple consiste à laisser le processus finir le `main` avec l'instruction `return` suivie du code de retour du programme. Une autre est de terminer le programme grâce à la fonction :

```
#include <stdlib.h>

void exit(status);
```

Celle-ci a pour avantage de quitter le programme quel que soit la fonction dans laquelle on se trouve.
Par exemple, avec ce code :

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

void quit(void)
{
    printf(" Nous sommes dans la fonction quit().\n");
    exit(EXIT_SUCCESS);
}

int main(void)
{
    quit();
    printf(" Nous sommes dans le main.\n");
    return EXIT_SUCCESS;
}
```

Le résultat est sans appel :

```
$ ./a.out
 Nous sommes dans la fonction quitter
```

L'exécution montre que l'instruction `printf("Nous sommes dans le main.\n");`n'est pas exécutée. Le programme s'est arrêté à la lecture de `exit(EXIT_SUCCESS);`.

Pour autant, cela ne vous dispense pas de mettre un `return` à la fin du main. Si vous ne le faites pas, vous risquez d'avoir un Warning à la compilation. Exemple, avec ce code :

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

void quit(void)
{
    printf(" Nous sommes dans la fonction quit().\n");
    exit(EXIT_SUCCESS);
}

int main(void)
{
    quit();
    printf(" Nous sommes dans le main.\n");
}
```

Qui produit ce Warning à la compilation :

```
$ gcc essai.c -Wall
essai.c: In function ‘main’:
essai.c:16: warning: control reaches end of non-void function
$
```

Ajoutez un petit `return EXIT_SUCCESS;` à la fin du main, ça ne coûte rien (et pas de `void main(void)` qui produit un `warning: return type of ‘main’ is not ‘int’` à la compilation).

1-4-2-Terminaison anormale d’un processus

Pour quitter, de manière propre, un programme, lorsqu'un bug a été détecté, on utilise la fonction :

```c
#include <stdlib.h>


void abort(void);
```

Un prototype simple pour une fonction qui possède un défaut majeur : il est difficile de savoir à quel endroit du programme le bug a eu lieu.
Pour y remédier, il est préférable d'utiliser la macro `assert`, déclarée qui fonctionne comme suit.

- Elle prend en argument une condition.
- Si cette condition est vraie,`assert` ne fait rien.
- Si elle est fausse, la macro écrit un message contenant la condition concernée, puis quitte le programme. Cela permet d'obtenir une bonne gestion des erreurs.

N'utilisez `assert` que dans les cas critiques, dans lesquels votre programme ne peut pas continuer si la condition est fausse.
Voici un exemple de son utilisation avec la fonction `fork`.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <assert.h>
#include <sys/types.h>

int main(void)
{
    pid_t pid_fils;

    do {
    pid_fils = fork();


    } while (pid_fils == -1 && (errno == EAGAIN));

    assert(pid_fils != -1);

    if (pid_fils == 0) {
    printf("\n Je suis le fils !\n");
    } else {
```

```
    printf("\n Je suis le père !\n");
    }

    return EXIT_SUCCESS;
}
```

Vous pouvez remarquer que cette utilisation ne sert pas à grand chose, car il est aussi facile et rapide d'utiliser un `if`s uivi d'un `perror` et d'un `exit`. Néanmoins, vous vous rendrez compte par la suite que, parfois,`assert` est très pratique. Par exemple, il permet de remplacer une suite d'instruction du style :

```
if (*n == NULL) {
    exit(EXIT_FAILURE);
}
```

en :

```
assert(*n != NULL);
```

1-4-3-Exécution de routines de terminaison

Grâce à la programmation système, il est possible d'exécuter automatiquement telle ou telle fonction au moment où le programme se termine normalement, c'est-à-dire à l'aide des instructions `exit` et `return`.
Pour cela, deux fonctions sont disponibles :`atexit` et `on_exit`.

- **atexit**

Voici le prototype de cette fonction :

```
#include <stdlib.h>

int atexit(void (*function) (void));
```

Le paramètre est un pointeur de fonction vers la fonction à exécuter lors de la terminaison. Elle renvoie **0** en cas de réussite ou **-1** sinon.

Vous pouvez également enregistrer plusieurs fonctions à la terminaison. Si c'est le cas, lors de la fin du programme, les fonctions mémorisées sont invoquées dans l'**ordre inverse** de l'enregistrement.

Par exemple, voici deux codes qui affichent respectivement « *Au revoir* » et « *1* », « *2* », « *3* » lors de la terminaison.

Premier code :

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

void routine(void)
{
    printf(" Au revoir !\n");
}

int main(void)
{
    if (atexit(routine) == -1) {
    perror("atexit :");
    return EXIT_FAILURE;
    }

    printf(" Allez, maintenant, on quitte ! :D\n");

    return EXIT_SUCCESS;
}
```

```
$ ./a.out
```

```
 Allez, maintenant, on quitte !
```

```
 Au revoir !
```

```
$
```

Deuxième code :

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

void routine1(void)
{
    printf(" 1\n");
}

void routine2(void)
{
    printf(" 2\n");
}

void routine3(void)
{
    printf(" 3\n");
}

int main(void)
{
    if (atexit(routine3) == -1) {
    perror("atexit :");
    return EXIT_FAILURE;
    }
```

```
    if (atexit(routine2) == -1) {
    perror("atexit :");
    return EXIT_FAILURE;
    }

    if (atexit(routine1) == -1) {
    perror("atexit :");
    return EXIT_FAILURE;
    }

    printf(" Allez, maintenant, on quitte ! :D\n");

    return EXIT_SUCCESS;


}
```

```
$ ./a.out
 Allez, maintenant, on quitte !
 1
 2
 3
$
```

- **on_exit**

La deuxième fonction qui permet d'effectuer une routine de terminaison est :

```
#include <stdlib.h>


int on_exit(void (*function) (int, void *), void *arg);
```

La fonction prend donc en paramètre deux arguments :

- un pointeur sur la fonction à exécuter, qui sera, cette fois, de la forme `void fonction(int code Retour, void* argument)`. Le premier paramètre de cette routine est un entier correspondant au code transmis avec l'utilisation de `return` ou de `exit`.
- L'argument à passer à la fonction.

Elle renvoie **0** en cas de réussite ou **-1** sinon.

Votre fonction de routine de terminaison recevra alors deux arguments : le premier est un `int` correspondant au code transmis à `return` ou à `exit` et le second est un pointeur générique correspondant à l'argument que l'on souhaitait faire parvenir à la routine grâce à `on_exit`.

Il est à noter qu'il est préférable d'utiliser `atexit` plutôt que `on_exit`, la première étant conforme C89, ce qui n'est pas le cas de la seconde.

1-4-3-Synchronisation entre père et fils

Lorsque le processus fils se termine avant le processus père, il devient un **zombie**. Pour permettre à un processus fils en état zombie de disparaître complètement, on utilise la fonction wait, qui se trouve dans la bibliothèque sys/wait.h, déclarée comme ceci :

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

un peu plus de détail sur son fonctionnement. Lorsque l'on appelle cette fonction, cette dernière bloque le processus à partir duquel elle a été appelée jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID de ce dernier. En cas d'erreur, la fonction renvoie la valeur **-1**.

On met quoi pour le paramètre de la fonction ?
Le paramètre **status** correspond au code de retour du processus. Autrement dit, la variable que l'on y passera aura la valeur du code de retour du processus (ce code de retour est généralement indiquée avec la fonction exit). Nous verrons, à la fin de cette sous-partie, comment interpréter cette valeur.

De manière plus précise, l'octet de poids faible est un code indiquant pourquoi le processus s'est arrêté et, si ce dernier a fait appel à exit, l'octet précédent contient le code de retour.
Et si on oublie de mettre wait, il se passe quoi ?

Si le processus père s'arrête sans avoir lu le code de retour de son fils, c'est un processus init(vous vous souvenez, le fameux processus au PID 1) qui va le faire afin de le libérer de cet état de zombie.

```
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));

    /* On retourne le PID du processus ainsi créé */
    return pid;
}
```

```c
/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
    printf(" Nous sommes dans le fils !\n"
       " Le PID du fils est %d.\n"
       " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

    if (wait(NULL) == -1) {
    perror("wait :");
    exit(EXIT_FAILURE);
    }
}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    child_process();
    break;
    /* Si on est dans le père */
    default:
    father_process(pid);
    break;
    }

    return EXIT_SUCCESS;
}
```

Attention : il faut mettre autant de `wait` qu'il y a de fils, car comme je l'ai expliqué dans le fonctionnement :

Lorsque l'on appelle cette fonction, cette dernière bloque le processus à partir duquel elle a été appelée jusqu'à ce qu'un de ses fils se termine.

.

- **Attendre la fin de n'importe quel processus**

Il existe également une fonction qui permet de suspendre l'exécution d'un processus père jusqu'à ce qu'un de ses fils, dont on doit passer le PID en paramètre, se termine. Il s'agit de la fonction **`waitpid`**:

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Plus précisément, la valeur de `pid` est interprétée comme suit :

- si *pid > 0*, le processus père est suspendu jusqu'à la fin d'un processus fils dont le PID est égal à la valeur *pid* ;
- si *pid = 0*, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils appartenant à son groupe ;
- si *pid = -1*, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils ;
- si *pid < -1*, le processus père est suspendu jusqu'à la mort de n'importe lequel de ses fils dont le GID est égal.

Le second argument, `status`, a le même rôle qu'avec `wait`.

Le troisième argument permet de préciser le comportement de waitpid. On peut utiliser deux constantes :

- WNOHANG : ne pas bloquer si aucun fils ne s'est terminé.
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Dans le cas où cela ne vous intéresse pas, il suffit de mettre le paramètre 0.

***Remarque** : Notez que `waitpid(-1, status, 0)` correspond à la fonction `wait`.*

1-4-4-Les macros status

Si l'on résume : un processus peut se terminer de façon normale (`return` ou `exit`) ou bien anormale (`assert`). Le processus existe toujours mais devient un zombie jusqu'à ce que `wait` soit appelé ou que le père meure.

Le code de retour du processus est stocké dans l'emplacement pointé par `status`.

Pour avoir toutes ces informations, nous pouvons utiliser les macros suivantes :

Macro	Description
`WIFEXITED(status)`	Elle renvoie vrai si le statut provient d'un processus fils qui s'est terminé en quittant le main avec`return`ou avec un appel à`exit`.
`WEXITSTATUS(status)`	Elle renvoie le code de retour du processus fils passé à`exit`ou à`return`. Cette macro est utilisable uniquement si vous avez utilisé`WIFEXITED`avant, et que cette dernière a renvoyé vrai.
`WIFSIGNALED(status)`	Elle renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause d'un signal.
`WTERMSIG(status)`	Elle renvoie la valeur du signal qui a provoqué la terminaison du processus fils. Cette macro est utilisable uniquement si vous avez utilisé `WIFSIGNALED` avant, et que cette dernière a renvoyé vrai.

Ces macros utilisent le `status`, et non un pointeur sur ce `status`. Ces quatre macros sont les plus utilisées et les plus utiles. Il en existe d'autres : pour plus d'infos, un petit `man 2 wait` devrait vous aider.

Un petit exemple pour mettre en application l'étude de ces macros ? Nous allons tester plusieurs codes.

Premier code :

```c
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* Pour faire simple, on déclare status en globale à la barbare */
int status;

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));
```

```
    /* On retourne le PID du processus ainsi créé */
    return pid;
}

/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
    printf(" Nous sommes dans le fils !\n"
       " Le PID du fils est %d.\n"
       " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

    if (wait(&status) == -1) {
        perror("wait :");
        exit(EXIT_FAILURE);
    }

    if (WIFEXITED(status)) {
    printf(" Terminaison normale du processus fils.\n"
           " Code de retour : %d.\n", WEXITSTATUS(status));
    }

    if (WIFSIGNALED(status)) {
    printf(" Terminaison anormale du processus fils.\n"
           " Tué par le signal : %d.\n", WTERMSIG(status));
    }

}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    child_process();
    break;
    /* Si on est dans le père */
    default:
    father_process(pid);
    break;
    }

    return EXIT_SUCCESS;
}
```

Résultat :

```
$ ./a.out
 Nous sommes dans le père
 Le PID du fils est 1510
 Le PID du père est 1508
```

```
 Nous sommes dans le fils
 Le PID du fils est 1510
 Le PPID du fils est 1508
 Terminaison normale du processus fils.
 Code de retour : 0.
$
```

Deuxième code :

```c
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));

    /* On retourne le PID du processus ainsi créé */
    return pid;
}

/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
    printf(" Nous sommes dans le fils !\n"
       " Le PID du fils est %d.\n"
       " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
    exit(EXIT_FAILURE);
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    int status;

    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

    if (wait(&status) == -1) {
        perror("wait :");
        exit(EXIT_FAILURE);
    }

    if (WIFEXITED(status)) {
    printf(" Terminaison normale du processus fils.\n"
           " Code de retour : %d.\n", WEXITSTATUS(status));
    }
```

```
    if (WIFSIGNALED(status)) {
    printf(" Terminaison anormale du processus fils.\n"
           " Tué par le signal : %d.\n", WTERMSIG(status));
    }

}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    child_process();
    break;
    /* Si on est dans le père */
    default:
    father_process(pid);
    break;
    }

    return EXIT_SUCCESS;
}
```

Résultat :

```
$ ./a.out
 Nous sommes dans le père
 Le PID du fils est 11433
 Le PID du père est 11431
 Nous sommes dans le fils
 Le PID du fils est 11433
 Le PPID du fils est 11431
 Terminaison normale du processus fils.
 Code de retour : 1.
$
```

Troisième code :

```
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
```

```
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));

    /* On retourne le PID du processus ainsi créé */
    return pid;
}

/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
    printf(" Nous sommes dans le fils !\n"
       " Le PID du fils est %d.\n"
       " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
    sleep(20);
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    int status;

    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

    if (wait(&status) == -1) {
        perror("wait :");
        exit(EXIT_FAILURE);
    }

    if (WIFEXITED(status)) {
    printf(" Terminaison normale du processus fils.\n"
           " Code de retour : %d.\n", WEXITSTATUS(status));
    }

    if (WIFSIGNALED(status)) {
    printf(" Terminaison anormale du processus fils.\n"
           " Tué par le signal : %d.\n", WTERMSIG(status));
    }

}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    child_process();
    break;
    /* Si on est dans le père */
    default:
    father_process(pid);
    break;
    }

    return EXIT_SUCCESS;
}
```

Résultat :

Dans un premier terminal (début de l'exécution) :

```
$ ./a.out
 Le PID du fils est 18745
 Le PID du père est 18743
 Nous sommes dans le fils
 Le PID du fils est 18745
 Le PPID du fils est 18743
```

Dans un second terminal :

```
$ kill 18745
$
```

Dans le premier terminal, fin de l'exécution :

```
$ ./a.out
 Le PID du fils est 18745
 Le PID du père est 18743
 Nous sommes dans le fils
 Le PID du fils est 18745
 Le PPID du fils est 18743
 Terminaison anormale du processus fils.
 Tué par le signal : 15.
$
```

Exercice.

Écrire un programme qui crée un fils. Le père doit afficher « Je suis le père » et le fils doit afficher « Je suis le fils ».

Correction :

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

int main(void)
{
    pid_t pid_fils;

    do {
    pid_fils = fork();
    } while ((pid_fils == -1) && (errno == EAGAIN));

    if (pid_fils == -1) {
    perror("fork");
    } else if (pid_fils == 0) {
    printf("Je suis le fils \n");
    } else {
    printf("Je suis le père \n");

    if (wait(NULL) == -1) {
        perror("wait :");
    }
    }

    return EXIT_SUCCESS;
}
```

2-Les fonctions de la famille exec

2-1- presentation et exemple

Dans cette sous-partie, vous allez pouvoir lancer des programmes. Pour cela, on doit vous présenter une famille de fonction nous permettant de réaliser cela : la famille `exec`. En réalité, il existe six fonctions appartenant à cette famille : : `execl`, `execle`, `execlp`, `execv`, `execve` et `execvp`. Parmi eux, seule la fonction `execve` est un appel-système, les autres sont implémentées à partir de celui-ci. Ces fonctions permettent de remplacer un programme en cours par un autre programme sans en changer le PID. Autrement dit, on peut remplacer le code source d'un programme par celui d'un autre programme en faisant appel à une fonction `exec`. Voici leurs prototypes :

```
int execve(const char *filename, char *const argv[], char *const envp[]);

int execl(const char *path, const char *arg,  ...);
```

```
int execlp(const char *file, const char *arg, ...);

int execle(const char *file, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);
```

- Suffixe en **-v** : les arguments sont passés sous forme de tableau ;
- Suffixe en **-l** : les arguments sont passés sous forme de liste ;
- Suffixe en **-p** : le fichier à exécuter est recherché à l'aide de la variable d'environnement **PATH** ;
- Pas de suffixe en **-p** : le fichier à exécuter est recherché relativement au répertoire de travail du processus père ;
- Suffixe en **-e** : un nouvel environnement est transmis au processus fils ;
- Pas de suffixe en **-e** : l'environnement du nouveau processus est déterminé à partir de la variable d'environnement externe `environ` du processus père.

Je vous explique rapidement à quoi correspond chaque caractéristique :

–

Le premier argument correspond au chemin complet d'un fichier objet exécutable (si `path`) ou le nom de ce fichier (si `file`).

– Le second argument correspond aux paramètres envoyés au fichier à exécuter : soit sous forme de liste de pointeurs sur des chaînes de caractères, soit sous forme de tableau. Le premier élément de la liste ou du tableau est le nom du fichier à exécuter, le dernier est un pointeur NULL.

–

Le troisième argument éventuel est une liste ou un tableau de pointeurs d'environnement.
De plus, toutes ces fonctions renvoient **-1**. `errno` peut correspondre à plusieurs constantes, dont EACCESS (vous n'avez pas les permissions nécessaires pour exécuter le programme), E2BIG (la liste d'argument est trop grande), ENOENT (le programme n'existe pas), ETXTBSY (le programme a été ouvert en écriture par

d'autres processus), ENOMEM (pas assez de mémoire), ENOEXEC (le fichier exécutable n'a pas le bon format) ou encore ENOTDIR (le chemin d'accès contient un nom de répertoire incorrect).

Comme dit auparavant, seul execve est un appel-système, et les autres ne sont que ses dérivés. La logique voudrait qu'on l'utilise « par défaut », mais on va plutôt utiliser la fonction execv qui fonctionne de la même manière que execve, mis à part que vous n'avez pas à vous soucier de l'environnement.

Maintenant,

<u>Application 1</u> : écrivez un programme qui lance la commande ps. Vous pouvez la trouver dans le dossier /bin.

Correction :

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

int main(void)
{
    /* Tableau de char contenant les arguments (là aucun : le nom du
       programme et NULL sont obligatoires)  */
    char *arguments[] = { "ps", NULL };

    /* Lancement de la commande */
    if (execv("/bin/ps", arguments) == -1) {
    perror("execv");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

```
$ ./a.out
PID TTY TIME CMD
1762 pts/0 00:00:00 bash
1845 pts/0 00:00:00 ps
```

Application 2 :

*Créez un programme qui prend en argument le chemin complet d'un répertoire et qui ouvre l'*analyseur d'utilisation des disques *pour ce chemin. La commande permettant de lancer ce programme se nomme baobab, elle prend en argument le chemin du répertoire devant être analysé et elle se situe dans le répertoire /usr/bin/baobab*.

```c
#include <stdio.h>


#include <stdlib.h>


#include <unistd.h>
```

```c
/* On récupère les arguments */

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

{

    /* On crée un tableau contenant le nom du programme, l'argument et le

       dernier "NULL" obligatoire  */

    char *arguments[] = { "baobab", argv[1], NULL };



    /* On lance le programme  */

    if (execv("/usr/bin/baobab", arguments) == -1) {

    perror("execv");

        return EXIT_FAILURE;

    }

    return EXIT_SUCCESS;

}
```

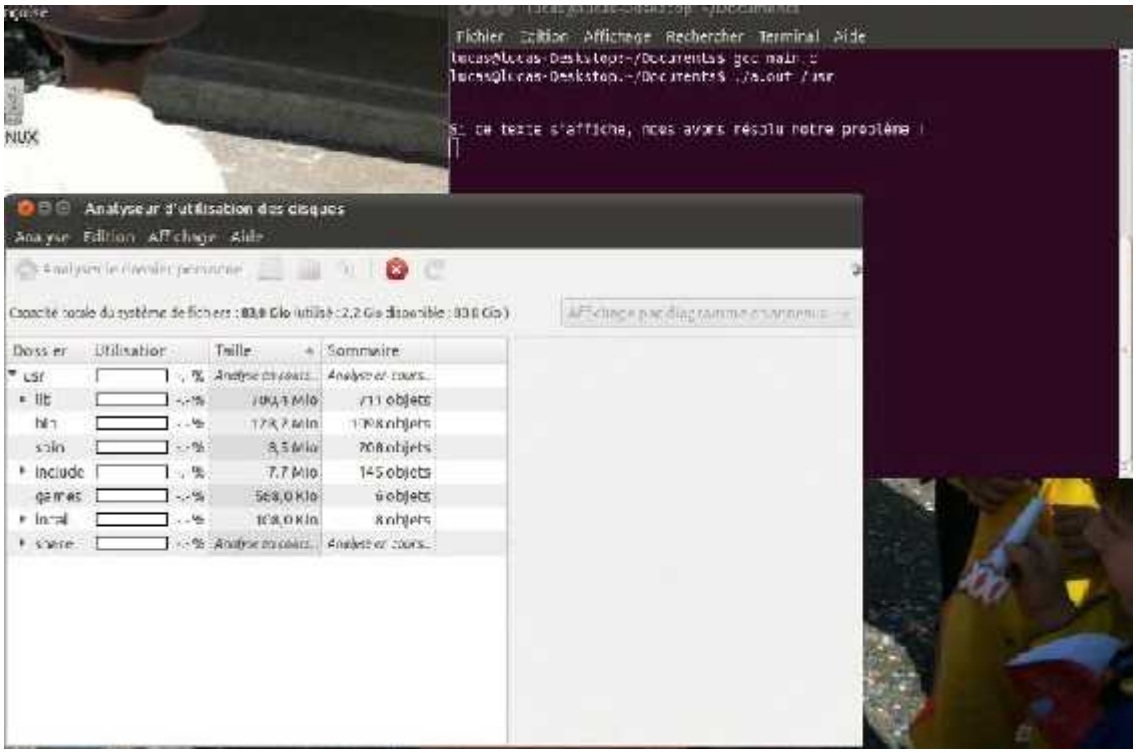
Maintenant, exécutez ce programme. Si vous avez tapé un chemin valide, l'*analyseur d'utilisation des disques* s'ouvre. Ensuite, ne fermez pas la fenêtre et retournez dans la console. Et là, stupeur ! On ne peut plus rentrer de commande, et s'il l'on fait CRTL + C, l'*AUD* se ferme ! Imaginez aussi que vous voulez exécuter un code à la suite de l'ouverture du programme...
Maintenant, réfléchissez. Il nous faut trouver une solution à ce problème : exécuter un nouveau programme tout en pouvant réaliser d'autres actions pendant ce temps. Vous avez compris ? Eh oui, il faut utiliser les processus !

Application 3 : Créez un programme qui règle notre problème...

```c
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
```

```c
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* La fonction create_process duplique le processus appelant et retourne
   le PID du processus fils ainsi créé */
pid_t create_process(void)
{
    /* On crée une nouvelle valeur de type pid_t */
    pid_t pid;

    /* On fork() tant que l'erreur est EAGAIN */
    do {
    pid = fork();
    } while ((pid == -1) && (errno == EAGAIN));

    /* On retourne le PID du processus ainsi créé */
    return pid;
}

/* La fonction son_process effectue les actions du processus fils */
void son_process(char *arg[])
{
    if (execv("/usr/bin/baobab", arg) == -1) {
    perror("execv");
    exit(EXIT_FAILURE);
    }
}

/* La fonction father_process effectue les actions du processus père */
void father_process(void)
{
    printf("\nSi ce texte s'affiche, nous avons résolu notre problème !\n");
}

int main(int argc, char *argv[])
{
    char *arg[] = { "baobab", argv[1], NULL };
    pid_t pid = create_process();

    switch (pid) {
    /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
    /* Si on est dans le fils */
    case 0:
    son_process(arg);
    break;
    /* Si on est dans le père */
    default:
    father_process();
    break;
    }

    return EXIT_SUCCESS;
}
```

Compilez, exécutez, résultat :



2-2-La fonction system

La fonction `system` est semblable aux `exec`, mais elle est beaucoup plus simple d'utilisation. En revanche, on ne peut pas y passer d'arguments. Son prototype est

```c
#include <stdlib.h>

int system(const char *command);
```

Écrivez un programme qui lance la commande *clear* qui permet d'effacer le contenu de la console.

```c
#include <stdio.h>

#include <stdlib.h>

int main(void)

{

    system("clear");

    return EXIT_SUCCESS;

}
```

C’est simple, mais il faut se méfier de la fonction `system`
En effet, la fonction `system` comporte des failles de sécurité **très importantes**.

Pour en savoir d’avantage , consulter le lien suivant :

https://www.cgsecurity.org/Articles/SecProg/Art1/index-fr.html