



La programmation système en C sous Unix

Par Lucas Pesenti (Lucas-84)



Sommaire

Sommaire	2
www.creationjeuxjava.fr Développement de jeux 2D ! La Communauté Française de Slick2D , la I	1
La programmation système en C sous Unix	3
Partie 1 : Les bases de la programmation système	3
Avant-propos	4
Introduction	4
Généralités sur le développement sous Unix	4
Conventions typographiques	5
Gestion des erreurs	5
Les processus	7
Présentation des processus	7
Parallélisme et pseudo-parallélisme	7
Programmes et processus	7
Espace d'adressage des processus	7
Ordonnancement des processus	8
Notions	8
Particularités de la gestion des processus sous Unix	8
PID	9
UID	9
GID	10
Organisation des processus	10
Les états d'un processus	10
Implémentation des processus	11
Commandes de gestion des processus	12
Création d'un nouveau processus	12
La fonction fork	12
Autres fonctions :	13
Code complet :	14
Le cast	15
Terminaison d'un processus	16
Terminaison d'un programme	16
Exécution de routines de terminaison	19
Synchronisation entre père et fils	21
Exécuter un programme	31
Notions pré-requises	31
Arguments en ligne de commande	31
La variable PATH	32
L'environnement	32
Les fonctions de la famille exec	35
La fonction system	39
Les threads	41
Qu'est ce qu'un thread ?	41
Compilation	41
Manipulation des threads	41
Créer un thread	41
Supprimer un thread	42
Première application	42
Attendre la fin d'un thread	43
Exercice résumé	43
Exclusions mutuelles	44
Problématique	44
Les mutex	44
Initialiser un mutex	44
Verrouiller un mutex	45
Déverrouiller un mutex	45
Détruire un mutex	45
Les conditions	45
Les tubes	48
Qu'est ce qu'un tube ?	49
Définition	49
Utilité	49
Vocabulaire	49
Manipuler les tubes	49
Créer	49
Écrire	49
Lire	50
Fermer	51
Pratique	52
Notions complémentaires sur les tubes	53
Entrées/Sorties et tubes	53
Tubes nommés	54



La programmation système en C sous Unix



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos [commentaires](#) pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.



Par

Lucas Pesenti (Lucas-84)

Mise à jour : 06/11/2012

Difficulté : Intermédiaire  Durée d'étude : 3 heures



1 562 visites depuis 7 jours, classé 89/797

Bienvenue à toutes et à tous !

Alors comme ça, vous êtes tentés par la *programmation système en C sous Unix* ?

Peut-être ne savez-vous pas ce que c'est, et avez cliqué par curiosité. C'est pourquoi, avant de commencer, nous allons répondre à la question :



Qu'est-ce que la programmation système ?

Lorsque l'on dispose d'un système d'exploitation, ce dernier permet de différencier deux types de programmes :

- les **programmes d'application** des utilisateurs. Ces programmes sont réalisés lors de la programmation dite « classique », celle que vous avez fait par exemple pendant le tutoriel sur le C de M@teo21.
- Les **programmes systèmes** qui permettent le fonctionnement de l'ordinateur. C'est ce type de programme que nous allons créer dans ce cours.

Le système d'exploitation sur lequel vous travaillerez devra faire partie de la « famille Unix », dont font partie GNU/Linux, Mac OS, Free BSD... Vous trouverez une schématisation plus ou moins exhaustive [ici](#).

Il faut savoir que le langage C, à partir duquel nous programmerons, a été créé spécialement pour la programmation système, plus précisément pour le développement du système d'exploitation... UNIX. Il est donc particulièrement adapté à ce type de programmation.

Le seul prérequis nécessaire à la lecture de ce cours est de connaître les bases du langage C. Les deux premières parties du [tutoriel C](#) du site seront suffisantes.

Avoir quelques connaissances sur les systèmes d'exploitation de type Unix est également préférable (bien que de nombreuses explications seront reprises dans ce cours).

Eh bien, fini les bavardages, on passe à la pratique !

Partie 1 : Les bases de la programmation système

Aux armes, zéroïens !

Nous commençons ce tutoriel par une étude des bases de la programmation système.

Au menu : processus, signaux, threads...

Que du bon quoi ! 😊

Avant-propos

Avant de nous jeter corps et âme dans la programmation système, commençons par étudier quelques notions théoriques sur la programmation système et sur la famille Unix.

A la fin de ce premier chapitre, vous saurez :

- définir précisément la programmation système ;
- le fonctionnement général de la programmation sous Unix ;
- gérer correctement vos erreurs.

Introduction



Quel est le but de ce tutoriel ?

Ce cours a pour but de vous apprendre à maîtriser toutes les finesses de la programmation système.



Qu'est-ce que nous allons savoir faire ?

La programmation système permet de *créer des drivers, communiquer avec les périphériques*, voire même *créer un système d'exploitation* !

(mais bon on en est pas encore là hein 😊)

Par exemple, les emblématiques [Apache](#), [Emacs](#), [gcc](#), [gdb](#) ou encore [glibc](#) sont des programmes systèmes.

Généralités sur le développement sous Unix

Dans cette troisième partie, nous allons aborder quelques termes de vocabulaire indispensables pour la suite du cours.

Les systèmes *Unix* sont des systèmes d'exploitation qui sont constitués de plusieurs programmes, et chacun d'eux fournit un service au système. Tous les programmes qui fournissent des services similaires sont regroupés dans une **couche logicielle**.

Une couche logicielle qui a accès au matériel informatique s'appelle une couche d'**abstraction matérielle**.

Le **noyau** est une sorte de logiciel d'arrière-plan qui assure les communications entre ces programmes. C'est donc par lui qu'il va nous falloir passer pour avoir accès aux informations du système.

Pour accéder à ces informations, nous allons utiliser des fonctions qui permettent de communiquer avec le noyau. Ces fonctions s'appellent des **appels-systèmes**.

De manière plus théorique, le terme « appel-système » désigne l'appel d'une fonction, qui, depuis l'espace utilisateur, demande des services ou des ressources au système d'exploitation. Par exemple, les fonctions `read` et `write` sont des appels-systèmes.

Chaque architecture matérielle ne supporte que sa propre liste d'appels-systèmes, c'est pourquoi les appels-systèmes diffèrent d'une machine à l'autre. Heureusement, la plupart d'entre eux (90% environ) sont implémentés sur toutes les machines. Les 10% restants ne seront pas utilisés dans ce cours. 😊

Pour des raisons de sécurité évidentes, les applications de l'espace utilisateur ne peuvent pas directement exécuter le code du noyau ou manipuler ses données. Par conséquent, un mécanisme de signaux a été mis en place. Quand une application exécute un appel-système, elle peut alors effectuer un **trap**, et peut exécuter le code, du moment que le noyau le lui autorise.

On peut également qualifier le système de **multitâche**, ce qui signifie qu'il est capable de gérer l'exécution de plusieurs programmes en simultané et de **multi-utilisateur** car il permet que plusieurs utilisateurs aient accès au même ordinateur en

même temps, et qu'ils puissent profiter des mêmes ressources.

Conventions typographiques

Derrière ce titre un peu barbare, je vais vous montrer comment ce cours va être administré pour mettre en valeur les expressions :

Noms propres, matériaux en italique

Exemple : Sous un environnement *GNOME* de *Linux*, quand j'ai ouvert l'*analyseur d'utilisation des disques*, rien ne s'est passé.

Police courrier pour les chemins ou noms de répertoire/fichiers

Exemple : Rendez-vous dans le répertoire `/usr/include` et ouvrez le fichier nommé `passwd`.

Mots importants en gras

Exemple : Nous allons utiliser un **appel-système** qui va dupliquer le **processus** appelant.

Constantes définies par le système en violet

Exemple : Il existe deux constantes pour cela : `STDIN_FILENO` et `STDOUT_FILENO`.

Gestion des erreurs

Avec la programmation système, nous allons étudier et manipuler tout ce qui touche à votre système d'exploitation.

Ainsi, nous devons 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 :

Code : C

```
#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, je vous conseille de taper la commande `man errno`. Une description de chaque erreur y est disponible (en anglais ! 🤖).

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 :

Code : C

```
#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 :

Code : C

```
if (fork() == -1) {  
    perror("fork");  
}
```

Eh bien voilà, ce premier chapitre est terminé.

Nous avons fait un tour d'horizon de ce qui nous attend, et comment l'aborder.

Dans le prochain chapitre, on entre enfin la programmation système, avec une étude des *processus*.

Les processus

Dans ce chapitre, nous allons apprendre à maîtriser les processus de bout en bout, de leur création jusqu'à leur terminaison.

Présentation des processus

Dans cette première sous-partie, nous allons découvrir la notion de processus de façon généraliste, c'est-à-dire les concepts de ces derniers présents dans tous les systèmes d'exploitation qui les utilisent.

Parallélisme et pseudo-parallélisme

La base des ordinateurs modernes, c'est le parallélisme. Il est désormais inconcevable qu'un ordinateur ne puisse exécuter plusieurs programmes en même temps.

Du point de vue matériel, le processeur passe d'un programme à un autre en quelques millisecondes, ce qui va donner à l'utilisateur une impression de simultanéité. C'est le **pseudo-parallélisme**, à différencier avec le véritable **parallélisme** des systèmes multiprocesseurs.

Les processus sont la base du pseudo-parallélisme.

Programmes et processus

Il est très important de différencier la notion de *programme* et la notion de *processus*.

La définition exacte d'un processus, c'est *programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire*. En effet, au cours de son exécution, les instructions d'un programme modifient les valeurs des registres (le compteur ordinal, le registre d'état...) ainsi que le contenu de la pile.

Un programme, c'est une suite d'instructions (notion **statique**), tandis qu'un processus, c'est l'image du contenu des registres et de la mémoire centrale (notion **dynamique**).

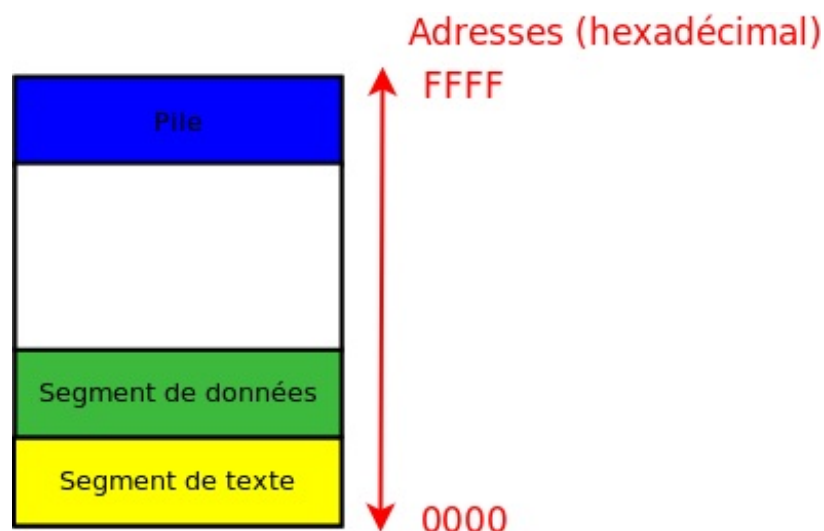
Espace d'adressage des processus

Chaque processus possède un **espace d'adressage**, c'est-à-dire un ensemble d'adresses mémoires dans lesquelles il peut lire et écrire.

Cet espace est divisé en trois parties :

- le **segment de texte** (le code du programme) ;
- le **segment de données** (les variables) ;
- la **pile**.

Voici une schématisation d'un espace d'adressage :

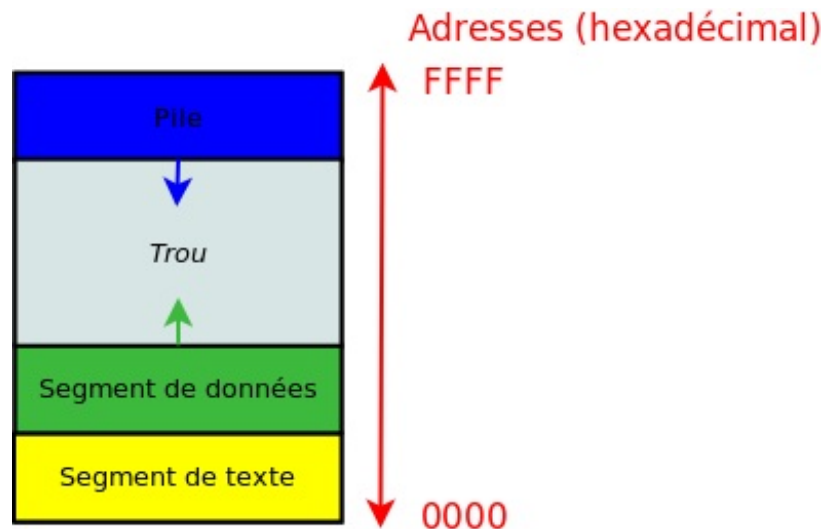


Hep, minute ! C'est quoi ce vide au milieu ?



Eh bien, c'est un trou. La pile empiète sur cet espace de manière automatique et l'extension du segment de données commence lors d'une allocation dynamique.

Le segment de texte (le code), quant à lui, ne bouge pas.



Ordonnancement des processus

Lorsqu'un processus est lancé, le système doit gérer la mémoire et l'allocation du processeur lui étant accordée. Il fait appel à l'**ordonnanceur** (*scheduler* en anglais).

Un système d'exploitation est **préemptif** lorsqu'il peut arrêter à tout moment n'importe quelle application pour passer à la suivante (exemple : Windows XP, Windows 7 et GNU/Linux sont des systèmes préemptifs). Il peut aussi être **coopératif** quand il permet à plusieurs applications de fonctionner et d'occuper la mémoire, et leur laissant le soin de gérer cette occupation (exemple : Windows 95, 98 et Millénium sont des systèmes coopératifs).

En résumé :

- **multitâche préemptif** : le système d'exploitation garde le contrôle et se réserve le droit de fermer l'application.
- **multitâche coopératif** : le système d'exploitation laisse les applications gérer.

Vous aurez donc compris que le multitâche coopératif est plus « dangereux » pour le système, car risque de blocage si une application fait des siennes. 😊

Enfin, dernière chose à retenir : **les systèmes basés sur Unix sont des systèmes préemptifs.**

Notions

Nous allons maintenant voir certaines notions et termes de vocabulaire relatifs aux processus, en étudiant leur implémentation sous Unix.

Particularités de la gestion des processus sous Unix

Dans les systèmes basés sur Unix tout particulièrement, les processus jouent un rôle très important. Le concept de processus a été mis au point dès les débuts de ce système : il a ainsi participé à sa gloire et à sa célébrité. Une des particularités de la gestion des processus sous Unix consiste à séparer la création d'un processus et l'exécution d'une image binaire. Bien que la plupart du temps ces deux tâches sont exécutées ensemble, cette division a permis de nouvelles libertés quant à la gestion des tâches. Par exemple, cela permet d'avoir plusieurs processus pour un même programme.

Autrement dit, sous les autres systèmes d'exploitation (mis à part quelques exceptions), processus = nouveau programme, alors que sous Unix ce n'est pas forcément le cas.

Ce principe, peu utilisé dans les autres systèmes, a survécu de nos jours. Alors que la plupart des systèmes d'exploitation offrent un seul appel-système pour exécuter un nouveau programme, Unix en possède deux : `fork` et `exec` (nous étudierons ce dernier dans le troisième chapitre).

PID

Chaque processus peut être identifié par son numéro de processus, ou **PID** (*Process Identifier*).

Un numéro de PID est unique dans le système : il est impossible que deux processus aient un même PID au même moment.

Lorsque l'on crée un processus (nous verrons comment faire dans la suite du chapitre), on utilise une fonction qui permet de dupliquer le processus appelant. On distingue alors les deux processus par leur PID. Le processus appelant est alors nommé **processus père** et le nouveau processus **processus fils**. Quant on s'occupe du processus fils, le PID du processus père est noté **PPID** (*Parent PID*).

Attribution du PID

Par défaut, le noyau attribue un PID avec une valeur inférieure à 32768. Le 32768ème processus créé reçoit la plus petite valeur de PID libérée par un processus mort entre-temps (paix à son âme... 🙏). Par ailleurs, cette valeur maximale peut être changée par l'administrateur en modifiant la valeur du fichier `/proc/sys/kernel/pid_max`.

De plus, les PID sont attribués de façon linéaire. Par exemple, si 17 est le PID le plus élevé affecté, un processus créé à cet instant aura comme PID 18. Le noyau réutilise les PID de processus n'existant plus uniquement quand la valeur de `pid_max` est atteinte.

UID

Les systèmes basés sur Unix sont particulièrement axés sur le côté multi-utilisateur. Ainsi, il existe de très nombreuses sécurités sur les permissions nécessaires pour exécuter telle ou telle action.

C'est pour cela que chaque utilisateur possède un identifiant, sous forme numérique, nommé **UID** (*User Identifier*).

En conséquence, nous pouvons également distinguer les processus entre eux par l'UID de l'utilisateur qui les a lancés.

Quelques remarques sur la valeur de l'UID

- La valeur de l'UID est comprise entre les constantes **UID_MIN** et **UID_MAX** du fichier `/etc/login.defs`.
- Conventionnellement, plus l'UID est bas, plus l'utilisateur a des privilèges. Néanmoins, l'attribution de l'UID est de la compétence de l'administrateur du système. Ainsi, cette règle n'est pas toujours vérifiable.
- Les valeurs inférieures à 100 sont généralement réservées aux utilisateurs standards (« par défaut » si vous préférez).

Différents UID

On distingue trois identifiants d'utilisateur par processus :

- l'UID **réel** : il correspond à l'UID de l'utilisateur qui a lancé le processus ;
- l'UID **effectif** : il correspond aux privilèges accordés à cet utilisateur ;
- l'UID **sauvé** : c'est la copie de l'ancien UID réel quand celui-ci a été modifié par un processus.

Pour examiner tous les utilisateurs de l'ordinateur, allez dans `/etc` et ouvrez le fichier texte nommé `passwd`.

Ce fichier rassemble des informations sur tous les utilisateurs ayant un compte sur le système ; une ligne par utilisateur et 7 champs par ligne, séparés par le caractère deux-points (:).

NOM : MOT DE PASSE : UID : GID : COMMENTAIRE : RÉPERTOIRE : COMMANDE

En fait, l'analyse de ce fichier ne nous importe peu mais cela peut vous permettre de connaître les différents utilisateurs de

l'ordinateur (les plus connus étant **root** et vous-même ; mais vous pouvez remarquer qu'il en existe plein d'autres!).

Permission Set – UID

Il existe une permission spéciale, uniquement pour les exécutable binaires, appelée la permission **Set – UID**. Cette permission permet à l'utilisateur ayant les droits d'exécution sur ce fichier d'exécuter le fichier avec les privilèges de son propriétaire. On met les droits Set - UID avec la commande `chmod` et l'argument `+s`. On passe en second argument le nom du fichier.

Exemple :

Code : Console

```
$ ls -l
total 4
-rw-r--r-- 1 lucas lucas 290 2010-12-01 15:39 zombie.sh
$ chmod +s zombie.sh
$ ls -l
total 4
-rwSr-Sr-- 1 lucas lucas 290 2010-12-01 15:39 zombie.sh
```

GID

Chaque utilisateur du système appartient à un ou plusieurs groupes. Ces derniers sont définis dans le fichier `/etc/groups`. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les UID, un processus dispose donc de plusieurs **GID (Group Identifier) réel, effectif, sauvé**, ainsi que de GID supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.

Connaître le GID d'un processus n'est pas capital, c'est pourquoi nous ne nous arrêterons pas sur ce point.

Organisation des processus

Les processus sont organisés en **hiérarchie**. Chaque processus doit être lancé par un autre (rappelez-vous les notions sur processus père et processus fils). La racine de cette hiérarchie est le **programme initial**. En voici quelques explications.

Le **processus inactif du système** (*System idle process* : le processus que le noyau exécute tant qu'il n'y a pas d'autres processus en cours d'exécution) a le PID 0. C'est celui-ci qui lance le premier processus que le noyau exécute, le **programme initial**. Généralement, sous les systèmes basés sous Unix, le programme initial se nomme **init**, et il a le PID 1.

Si l'utilisateur indique au noyau le programme initial à exécuter, celui-ci tente alors de le faire avec quatre exécutable, dans l'ordre suivant : `/sbin/init`, `/etc/init` puis `/bin/init`.

Le premier de ces processus qui existe est exécuté en tant que programme initial.

Si les quatre programmes n'ont pas pu être exécutés, le système s'arrête : **panique du noyau**...

Après son chargement, le programme initial gère le reste du démarrage : initialisation du système, lancement d'un programme de connexion... Il va également se charger de lancer les démons. Un **démon** (du terme anglais *daemon*) est un processus qui est constamment en activité et fournit des services au système.

Les états d'un processus

Un processus peut avoir plusieurs états :

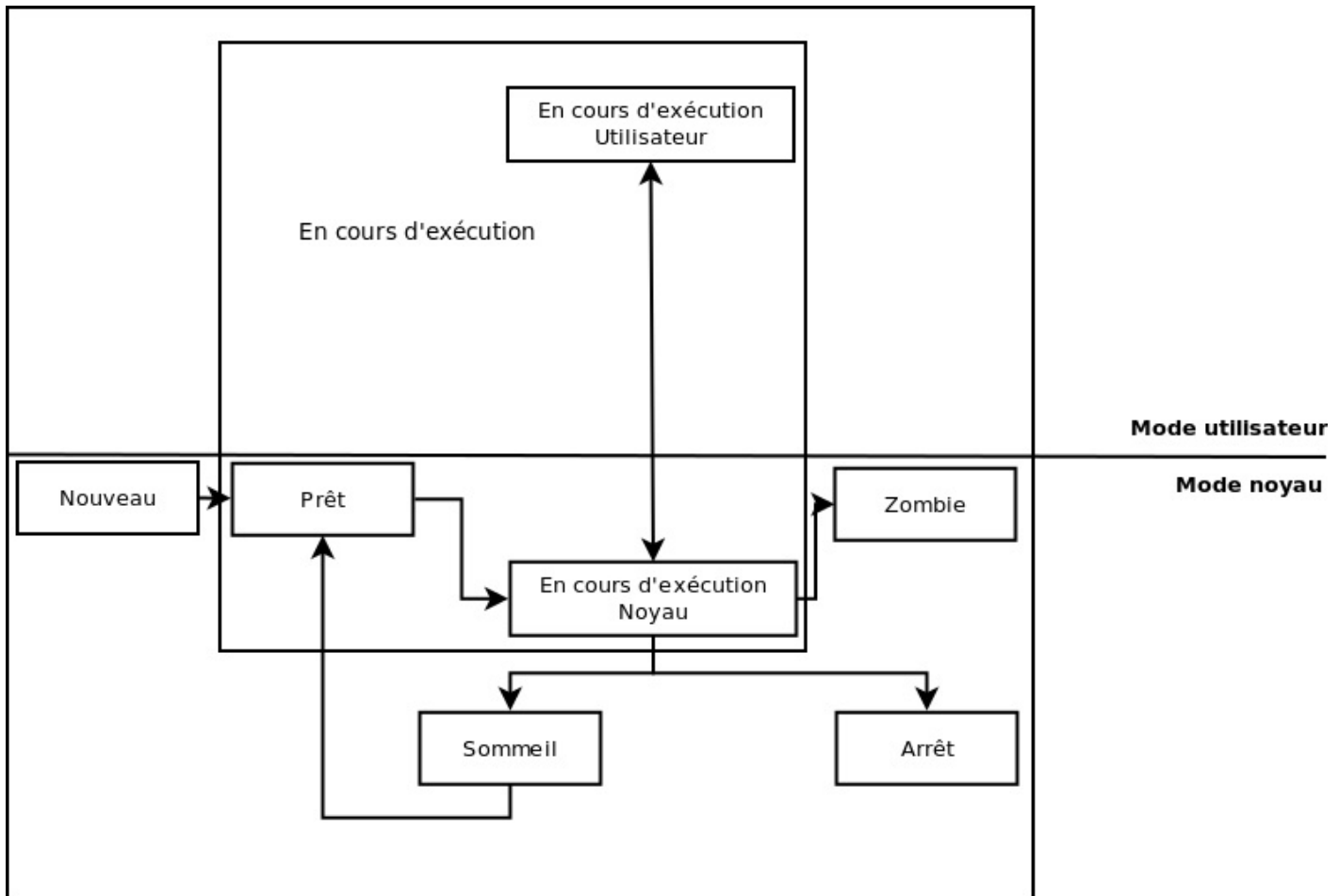
- **exécution (R pour *running*)** : le processus est en cours d'exécution ;
- **sommeil (S pour *sleeping*)** : dans un multitâche coopératif, quand il rend la main ; ou dans un multitâche préemptif, quand il est interrompu au bout d'un quantum de temps ;
- **arrêt (T pour *stopped*)** : le processus a été temporairement arrêté par un signal. Il ne s'exécute plus et ne réagira qu'à un

signal de redémarrage ;

- **zombie** (**Z** pour ... *zombie*) : le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

De plus, sous Unix, un processus peut évoluer dans deux modes différents : le mode **noyau** et le mode **utilisateur**. Généralement, un processus utilisateur entre dans le mode noyau quand il effectue un appel-système.

En guise de résumé, voici un diagramme d'états des processus sous Unix.



Implémentation des processus

Pour implémenter les processus, le système d'exploitation utilise un tableau de structure, appelé **table des processus**. Cette dernière comprend une entrée par processus, allouée dynamiquement, correspondant au processus associé à ce programme : c'est le **bloc de contrôle du processus** (*Process Control Block*, souvent abrégé **PCB**). Ce bloc contient, entre autres, les informations suivantes :

- le PID, le PPID, l'UID et le GID du processus ;
- l'état du processus ;
- les fichiers ouverts par le processus ;
- le répertoire courant du processus ;
- le terminal attaché au processus ;
- les signaux reçus par le processus ;
- le contexte processeur et mémoire du processus (c'est-à-dire l'état des registres et des données mémoires du processus).

Grâce à ces informations stockées dans la table des processus, un processus bloqué pourra redémarrer ultérieurement avec les mêmes caractéristiques.

Commandes de gestion des processus

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

Code : Console

```
$ 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 ; y : démons) :

Code : Console

```
$ 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).

Création d'un nouveau processus

Bon maintenant, nous allons pouvoir passer à la pratique !

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



Code : C

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

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**).

Que fait un **fork** au niveau du système d'exploitation ?

Lors de l'exécution de l'appel-système **fork**, le noyau effectue les opérations suivantes :

- il alloue un bloc de contrôle dans la table des processus.
- il copie les informations contenues dans le bloc de contrôle du père dans celui du fils sauf les identificateurs (PID, PPID...).
- il alloue un PID au processus fils.
- il associe au processus fils un segment de texte dans son espace d'adressage. Le segment de données et la pile ne lui

seront attribués uniquement lorsque celui-ci tentera de les modifier. Cette technique, nommée « *copie on write* », permet de réduire le temps de création du processus.

- l'état du processus est mis à l'état *exécution*.



Au passage, je vous propose un lien expliquant autrement ce que fait un `fork` : [cliquez ici](#).

Bref, revenons à la programmation. La fonction `fork` retourne une valeur de type `pid_t`. Il s'agit généralement d'un `int` ; il est 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` :

Code : C

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

pid_t fork(void);
```

Autres fonctions :

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

Code : C

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

pid_t getpid(void);
```

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

Code : C

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

pid_t getppid(void);
```

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

Code : C

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

uid_t getuid(void);
```

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

Code : C

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

gid_t getgid(void);
```



En passant, si vous avez déjà fait de l'anglais au moins une fois dans votre vie, vous devriez savoir que *get* peut se traduire par *obtenir*. C'est un bon moyen mnémotechnique pour retenir ces deux fonctions (`getpid` : *obtenir PID*, `getppid` : *obtenir PPID*, `getuid` : *obtenir UID* et `getgid` : *obtenir GID*).

Code complet :

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

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>

/* 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 :

Code : Console

```
$ ./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
```



En réalité, il est impossible de savoir exactement quel résultat on va avoir quand plusieurs processus exécutent leur code en même temps dans un même terminal.

Le cast

Ceci vous a peut-être surpris :

Code : C

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

Cette manipulation, bien que très utilisée dans le monde de la programmation, est étrangement peu connue dans les rangs de ce site.

Même si ce n'est pas vraiment le but du tutoriel, je vais y revenir dessus, car nous l'utiliserons assez souvent.

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 :

Code : C

```
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`.



Je finis sur un dernier *Warning* avant de nous quitter pour ce deuxième chapitre : attention à ne pas placer `fork` dans des boucles infinies (évitiez de le mettre dans une boucle tout court d'ailleurs...). En effet, il n'y a aucune sécurité sur le système : imaginez un peu le désastre si vous vous créez des milliers de processus...

Terminaison d'un processus

Terminaison d'un programme

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

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 :

Code : C

```
#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 :

Code : C

```
#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 :


Code : Console

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

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 :

Code : C



```
#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 :

Code : Console

```
$ 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).

Terminaison anormale d'un processus

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

Code : 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`.

Code : 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` suivi 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 :

Code : C

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

en :

Code : C

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

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 :

Code : C

```
#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 :

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;
}
```

Code : Console

```
$ ./a.out
Allez, maintenant, on quitte ! :D
Au revoir !
$
```

Deuxième code :**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;
}
```

Code : Console

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

on_exit

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

Code : C

```
#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 codeRetour, 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.

Synchronisation entre père et fils

Gare aux zombies !

Lorsque le processus fils se termine avant le processus père, il devient un **zombie** (pour ceux à qui ce terme ne dit rien, remontez un peu plus haut dans la page... 🤪). 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 :

Code : C

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

pid_t wait(int *status);
```

Je vais détailler un petit peu 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**.



Hep, minute ! 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 j'oublie de mettre `wait`, il se passe quoi ?

~~Votre ordinateur explose~~ 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.

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());
}

/* 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 :



Citation : Le cours

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** :

Code : C

```

#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.





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

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

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 :**Code : Console**

```

$ ./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 :**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 :**Code : Console**

```

$ ./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 :

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());
    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) :

Code : Console

```
$ ./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 :

Code : Console

```
$ kill 18745
$
```

Dans le premier terminal, fin de l'exécution :

Code : Console

```
$ ./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.
$
```

Très facile, le QCM, n'est-ce pas ? 😊

Bon, pour clore ce chapitre, voici un exercice très facile que vous pouvez réaliser si vous avez bien suivi ce chapitre.

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 :

Secret (cliquez pour afficher)

Code : 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");
    } else {
        printf("Je suis le père");

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

    return EXIT_SUCCESS;
}
```

Vous pouvez aussi vous amuser à créer plusieurs processus, qui exécutent chacun une tâche spécifique.

Bon, nous avons appris pleins de choses, mais les processus c'est pas très impressionnant n'est ce pas ?

C'est ce que nous allons voir dans le prochain chapitre : on va apprendre à lancer un programme à partir de notre programme bien aimé. 😊

Exécuter un programme

Voici un chapitre (enfin !) intéressant. Nous allons apprendre à exécuter un programme externe à partir de *notre* programme.

Notions pré-requises

Arguments en ligne de commande

La fonction `main` d'un programme peut prendre des arguments en ligne de commande. En effet, voici un exemple :

Code : Console

```
$ ./a.out argument1 argument2 argument3
```

Pour récupérer les arguments dans un programme en C, on utilise les paramètres `argc` et `argv` de la fonction `main`. Souvenez-vous de vos premiers cours de C en compagnie de M@teo21, ce dernier vous harassant à utiliser un prototype de `main` incompréhensible et impossible à retenir ! Voilà à quoi ces paramètres servent...

- `Argc` est un entier de type `int` qui donne le nombre d'arguments passés en ligne de commande **plus 1**.
- `Argv` est un tableau de pointeurs. `Argv[0]` contient le nom du fichier exécutable du programme. Les cases suivantes `argv[1]`, `argv[2]`, etc. contiennent les arguments passés en ligne de commande. Enfin, `argv[argc]` doit obligatoirement être `NULL`.

Fort de ces indications, écrivez un programme qui prend des arguments et qui affiche :

Code : Console

```
Argument 1 : ...  
Argument 2 : ...  
etc.
```

C'est bon ?

Correction !

Code : C

```
#include <stdio.h>  
#include <stdlib.h>  
  
/* Les deux paramètres de la fonction main() pour récupérer les  
arguments de la ligne de commande */  
int main(int argc, char *argv[])  
{  
    /* Déclaration de la variable pour parcourir le tableau argv */  
    int i;  
    /* Si le programme n'a pas reçu d'argument (rappelez vous argc  
est  
égal au nombre d'argument + 1) */  
    if (argc <= 1) {  
        printf("\nLe programme n'a reçu aucun argument\n");  
    }  
  
    /* Si il en a reçu un ou plus */  
    else {  
        for (i = 1; i < argc; i++) {  
            printf("Argument %d : %s\n", i, argv[i]);  
        }  
    }  
  
    return EXIT_SUCCESS;  
}
```

```
}
```

Résultat :

Code : Console

```
$ ./a.out je mange de la choucroute  
Argument 1 : je  
Argument 2 : mange  
Argument 3 : de  
Argument 4 : la  
Argument 5 : choucroute  
$
```

La variable PATH

La variable **PATH** contient une série de chemin vers des répertoires qui contiennent des exécutables ou des scripts de commande.

Comme toute variable qui se respecte, on peut afficher sa valeur avec la commande :

Code : Console

```
$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Lorsqu'on lance une commande dans la console, le système va chercher l'exécutable dans les chemins donnés dans le **PATH**. Pour le **PATH** donné en exemple, le noyau ira chercher l'exécutable dans `/usr/local/sbin`, puis dans `/usr/local/bin`, etc... En conséquence, si deux commandes portent le même nom, c'est la première trouvée qui sera exécutée.

L'environnement

Une application peut être exécutée dans des contextes différents : terminaux, répertoire de travail...

C'est pourquoi le programmeur système a souvent besoin d'accéder à l'**environnement**. Celui-ci est défini sous la forme de **variables d'environnement**.

Variables d'environnement

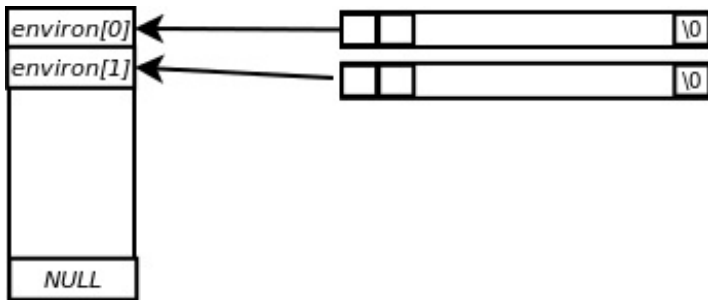
Elles sont définies sous la forme suivante : **NOM=VALEUR**.

Lorsqu'un programme en C démarre, ces variables sont automatiquement copiées dans un tableau de `char`. Vous pouvez y accéder en déclarant la variable externe globale **environ** au début de votre fichier, comme ceci :

Code : C

```
extern char **environ;
```

Ce tableau contient des chaînes de caractères se terminant par **NULL**, et lui-même se termine par un pointeur nul. Un petit schéma peut-être ?



Bien, maintenant, on va écrire un petit programme qui affiche toutes les variables d'environnement.

Code : C

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

extern char **environ;

int main(void)
{
    int i;

    for (i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }

    return EXIT_SUCCESS;
}
```

Vœici un exemple de résultat :

Code : Console

```
ORBIT_SOCKETDIR=/tmp/orbit-lucas
SSH_AGENT_PID=1412
GPG_AGENT_INFO=/tmp/gpg-oRe8EV/S.gpg-agent:1413:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=66803415d39a03f0d150db3d000000f-1300522273.896593-1253931584
WINDOWID=119537667
GNOME_KEYRING_CONTROL=/tmp/keyring-noSBVu
GTK_MODULES=canberra-gtk-module:gail:atk-bridge
USER=lucas
SSH_AUTH_SOCK=/tmp/keyring-noSBVu/ssh
SESSION_MANAGER=local/lucas-Desktop:@/tmp/.ICE-unix/1352,unix/lucas-Desktop:/tmp/.ICE-unix/1352
USERNAME=lucas
DEFAULTS_PATH=/usr/share/gconf/gnome.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-gnome:/etc/xdg
DESKTOP_SESSION=gnome
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/lucas/Documents
GDM_KEYBOARD_LAYOUT=fr oss
LANG=fr_FR.UTF-8
MANDATORY_PATH=/usr/share/gconf/gnome.mandatory.path
GDM_LANG=fr_FR.UTF-8
GDMSESSION=gnome
SHLVL=1
HOME=/home/lucas
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=lucas
```

```
XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
XTApH2KvXM,guid=7d07bc72de6da00b353c54e400000013
LESSOPEN=| /usr/bin/lesspipe %s
WINDOWPATH=7
DISPLAY=:0.0
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/var/run/gdm/auth-for-lucas-N38PVj/database
COLORTERM=gnome-terminal
OLDPWD=/home/lucas
_=./a.out
```

Variables d'environnement classiques

Voici une explication des variables d'environnement couramment utilisées :

- **HOME** : contient le répertoire personnel de l'utilisateur ;
- **PATH** : Tiens donc ! Comme on se retrouve... 🤪 ;
- **PWD** : contient le répertoire de travail ;
- **USER** (ou **LOGNAME**) : nom de l'utilisateur ;
- **TERM** : type de terminal utilisé ;
- **SHELL** : shell de connexion utilisé.

Créer, modifier, rechercher et supprimer une variable

La fonction :

Code : C

```
int putenv(const char *string);
```

sert à créer une variable d'environnement. Elle prend en argument une chaîne de caractère du type « NOM=VALEUR ».

La fonction :

Code : C

```
int setenv(const char *name, const char *value, int overwrite);
```

sert à modifier une variable d'environnement. Elle prend en argument le nom de la variable, la valeur à affecter et si on écrase la précédente valeur de la variable (si il y en a une) ou pas (1 pour l'écraser, 0 sinon).

La fonction `getenv`, ainsi déclarée dans `stdlib.h` :

Code : C

```
char *getenv(const char *name);
```

permet de rechercher une variable d'environnement.

Enfin, la fonction :

Code : C

```
void unsetenv(const char *name);
```

permet de supprimer une variable d'environnement.

Créez un programme permettant de rechercher les variables d'environnement passées en paramètre de la fonction `main`.

Code : C

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

int main(int argc, char *argv[])
{
    int i;
    char *variable;

    if (argc <= 1) {
        fprintf(stderr, "Le programme n'a reçu aucun argument.\n");
        return EXIT_FAILURE;
    }

    for (i = 1; i < argc; i++) {
        variable = getenv(argv[i]);

        if (!variable) {
            printf("%s : n'existe pas\n", argv[i]);
        } else {
            printf("%s : %s\n", argv[i], variable);
        }
    }

    return EXIT_SUCCESS;
}
```

Exemple d'exécution :

Code : Console

```
$ ./a.out DISPLAY HOME
DISPLAY : :0.0
HOME : /home/lucas
```

Les fonctions de la famille `exec`

Bien, nous allons maintenant rentrer dans le vif du sujet de ce chapitre : dans cette sous-partie, vous allez pouvoir lancer des programmes.

Pour cela, je dois 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 :

Code : C

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

Code : C

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

Code : C

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

Code : C

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

Code : C

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

Code : C

```
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 je l'ai dit tout à l'heure, seul **execve** est un appel-système, et les autres ne sont que ses dérivés. La logique de mon sadisme (🐱) voudrait qu'on l'utilise « par défaut », mais je vous conseille plutôt 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, écrivez un programme qui lance la commande `ps`. Vous pouvez la trouver dans le dossier `/bin`.

Correction :

Code : 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;
}
```

Code : Console

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

Maintenant que vous pouvez remarquer que ça a marché, 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`.

Code : C

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

/* 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 CTRL + C, l'*LAUD* 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 !

Créez un programme qui règle notre problème...

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 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) {
```

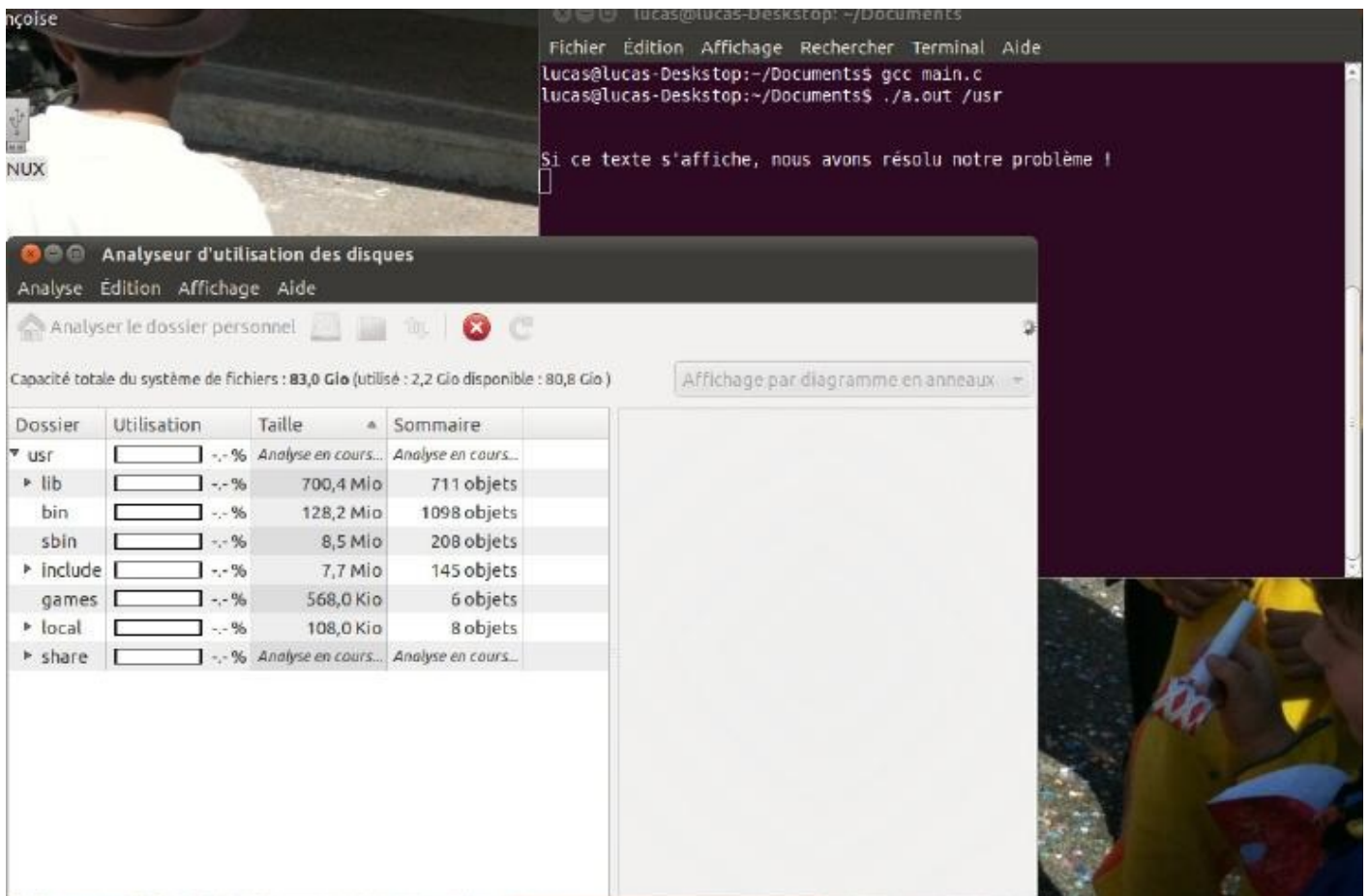
```

    /* 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 :



C'est gagné ! C'est gagné !

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

Code : 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.

Code : C

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

int main(void)
{
    system("clear");
    return EXIT_SUCCESS;
}
```

Vous avez vu comme c'est simple ? 😊

Mais c'est trop beau pour être vrai !

En effet, la fonction `system` comporte des failles de sécurité **très importantes**.

Nous devons donc nous fixer la règle suivante :

Il ne faut jamais utiliser la fonction `system`



Pour en savoir plus sur les failles de sécurité de `system`, [cliquez ici](#).

Pffiiou !

Voilà le chapitre enfin terminé ! 😊

Il a été riche en découverte, et vous pouvez enfin créer des programmes à peu près intéressants...

Prochain chapitre en vu : les *threads*.

Au programme : exécuter des fonctions en même temps...

Les threads

Voici un chapitre à ne manquer sous aucun prétexte car c'est une des bases de la programmation système sous Unix : l'étude des *threads Posix* (*Posix* est le nom d'une famille de standards qui indique un code standard).

Qu'est ce qu'un thread ?

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un **thread** de contrôle unique, le **thread principal**. Du point de vue programmation, ce dernier exécute le `main`.

Vous avez pu remarquer, lors de notre étude des processus, qu'en général, le système réserve un processus à chaque application, sauf quelques exceptions. Beaucoup de programmes exécutent plusieurs activités en parallèle, du moins en apparent parallélisme, comme nous l'avons vu précédemment. Comme à l'échelle des processus, certaines de ces activités peuvent se bloquer, et ainsi réserver ce blocage à un seul thread séquentiel, permettant par conséquent de ne pas stopper toute l'application. Ensuite, il faut savoir que le principal avantage des threads par rapport aux processus, c'est la facilité et la rapidité de leur création. En effet, tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Cela évite donc l'allocation de tous ces espaces lors de la création, et il est à noter que, sur de nombreux systèmes, la création d'un thread est environ cent fois plus rapide que celle d'un processus.

Au-delà de la création, la superposition de l'exécution des activités dans une même application permet une importante accélération quant au fonctionnement de cette dernière.

Sachez également que la communication entre les threads est plus aisée que celle entre les processus, pour lesquels on doit utiliser des notions compliquées comme les tubes (voir chapitre suivant).



Le mot « *thread* » est un terme anglais qui peut se traduire par « *fil d'exécution* ». L'appellation de « *processus léger* » est également utilisée.

Compilation

Toutes les fonctions relatives aux *threads* sont incluses dans le fichier d'en-tête `<pthread.h>` et dans la bibliothèque `libpthread.a` (soit `-lpthread` à la compilation).

Exemple :

Écrivez la ligne de commande qui vous permet de compiler votre programme sur les *threads* constitué d'un seul fichier `main.c` et avoir en sortie un exécutable nommé `monProgramme`.

Correction :

Code : Console

```
gcc -lpthread main.c -o monProgramme
```

Et n'oubliez pas d'ajouter `#include <pthread.h>` au début de vos fichiers.

Manipulation des threads

Créer un thread

Pour créer un thread, il faut déjà déclarer une variable le représentant. Celle-ci sera de type `pthread_t` (qui est, sur la plupart des systèmes, un **typedef** d'`unsigned long int`).

Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

Code : C

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                  void *(*start_routine) (void *), void *arg);
```

Ce prototype est un peu compliqué, c'est pourquoi nous allons récapituler ensemble.

- La fonction renvoie une valeur de type `int` : 0 si la création a été réussie ou une autre valeur si il y a eu une erreur.
- Le premier argument est un pointeur vers l'identifiant du thread (valeur de type `pthread_t`).
- Le second argument désigne les attributs du thread. Vous pouvez choisir de mettre le thread en état joignable (par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...). Dans nos exemple, on mettra généralement `NULL`.
- Le troisième argument est un pointeur vers la fonction à exécuter dans le thread. Cette dernière devra être de la forme `void *fonction(void* arg)` et contiendra le code à exécuter par le thread.
- Enfin, le quatrième et dernier argument est l'argument à passer au thread.

Supprimer un thread

Et qui dit créer dit supprimer à la fin de l'utilisation.
Cette fois, ce n'est pas une fonction casse-tête :

Code : C

```
#include <pthread.h>

void pthread_exit(void *ret);
```

Elle prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

Première application

Voyons un premier code qui réutilise toutes les notions des threads que nous avons vu jusque là.

Code : C

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

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");

    return EXIT_SUCCESS;
}
```

```
}
```

On compile, on exécute. Et là, zut... Le résultat, dans le meilleur des cas, affiche le message de thread en dernier. Dans le pire des cas, celui-ci ne s'affiche même pas (ce qui veut dire que le **return** s'est exécuté avant le thread...). Ce qui normal, puisqu'en théorie, comme avec les processus, le thread principal ne va pas attendre de lui-même que le thread se termine avant d'exécuter le reste de son code.

Par conséquent, il va falloir lui en faire la demande. 🤖 Pour cela, ~~Dieu~~ pthread a créé la fonction `pthread_join`.

Attendre la fin d'un thread

Voici le prototype de cette fameuse fonction :

Code : C

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Elle prend donc en paramètre l'identifiant du thread et son second paramètre, un pointeur, permet de récupérer la valeur retournée par la fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de `pthread_exit`).

Exercice résumé

Écrivez un programme qui crée un thread demandant un nombre à l'utilisateur, l'incrémentant une fois puis l'affichant dans le main.

Code : C

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

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if (pthread_create(&thread1, NULL, thread_1, NULL)) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }

    if (pthread_join(thread1, NULL)) {
        perror("pthread_join");
        return EXIT_FAILURE;
    }
}
```

```
printf("Après la création du thread.\n");  
  
return EXIT_SUCCESS;  
}
```

Et le résultat est enfin uniforme :

Code : Console

```
Avant la création du thread.  
Nous sommes dans le thread.  
Après la création du thread.
```

Exclusions mutuelles Problématique

Avec les threads, toutes les variables sont partagées : c'est la **mémoire partagée**.

Mais cela pose des problèmes. En effet, quand deux threads cherchent à modifier deux variables en même temps, que se passe-t-il ? Et si un thread lit une variable quand un autre thread la modifie ?

C'est assez problématique. Par conséquent, nous allons voir un mécanisme de synchronisation : les **mutex**, un des outils permettant l'**exclusion mutuelle**.

Les mutex

Concrètement, un **mutex** est en C une variable de type `pthread_mutex_t`. Elle va nous servir de verrou, pour nous permettre de protéger des données. Ce verrou peut donc prendre deux états : **disponible** et **verrouillé**.

Quand un thread a accès à une variable protégée par un mutex, on dit qu'il *tient le mutex*. Bien évidemment, il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

Le problème, c'est qu'il faut que le mutex soit accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale. Mais nous ne sommes pas des barbares ! 🤖 Par conséquent, j'ai choisi de vous montrer une autre solution : déclarer le mutex dans une structure avec la donnée à protéger.

Allez, un petit exemple ne vous fera pas de mal :

Code : C

```
typedef struct data {  
    int var;  
    pthread_mutex_t mutex;  
} data;
```

Ainsi, nous pourrions passer la structure en paramètre à nos threads, grâce à la fonction `pthread_create`. 🤔

Passons à la pratique : comment manipuler les mutex grâce à pthread ?

Initialiser un mutex

Conventionnellement, on initialise un mutex avec la valeur de la constante `PTHREAD_MUTEX_INITIALIZER`, déclarée dans `pthread.h`.

Code : C

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

```
#include <pthread.h>

typedef struct data {
    int var;
    pthread_mutex_t mutex;
} data;

int main(void)
{
    data new_data;

    new_data.mutex = PTHREAD_MUTEX_INITIALIZER;

    return EXIT_SUCCESS;
}
```

Verrouiller un mutex

L'étape suivante consiste à établir une **zone critique**, c'est-à-dire la zone où plusieurs threads ont l'occasion de modifier ou de lire une même variable en même temps.

Une fois cela fait, on verrouille le mutex grâce à la fonction :

Code : C

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mut);
```

Déverrouiller un mutex

A la fin de la zone critique, il suffit de déverrouiller le mutex.

Code : C

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mut);
```

Détruire un mutex

Une fois le travail du mutex terminé, on peut le détruire :

Code : C

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mut);
```

Les conditions

Lorsqu'un *thread* doit patienter jusqu'à ce qu'un événement survienne dans un autre *thread*, on emploie une technique appelée la **condition**.

Quand un *thread* est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre *thread*.

Comme avec les *mutex*, on déclare la condition en variable globale, de cette manière :

Code : C

```
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un *mutex* :

Code : C

```
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t  
*nomMutex);
```

Pour réveiller un *thread* en attente d'une condition, on utilise la fonction :

Code : C

```
int pthread_cond_signal(pthread_cond_t *nomCondition);
```

Exemple :

Créez un code qui crée deux *threads* : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

Correction :

Code : C

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

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de
la condition */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du
mutex */

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
    pthread_t monThreadCompteur;
    pthread_t monThreadAlarme;

    pthread_create (&monThreadCompteur, NULL, threadCompteur,
(void*)NULL);
    pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL);
    /* Création des threads */

    pthread_join (monThreadCompteur, NULL);
    pthread_join (monThreadAlarme, NULL); /* Attente de la fin des
threads */

    return 0;
```

```

}

void* threadCompteur (void* arg)
{
    int compteur = 0, nombre = 0;

    srand(time(NULL));

    while(1) /* Boucle infinie */
    {
        nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
        compteur += nombre; /* On ajoute ce nombre à la variable compteur */

        printf("\n%d", compteur);

        if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
        {
            pthread_mutex_lock (&mutex); /* On verrouille le mutex */
            pthread_cond_signal (&condition); /* On délivre le signal :
condition remplie */
            pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

            compteur = 0; /* On remet la variable compteur à 0 */
        }

        sleep (1); /* On laisse 1 seconde de repos */
    }

    pthread_exit(NULL); /* Fin du thread */
}

void* threadAlarme (void* arg)
{
    while(1) /* Boucle infinie */
    {
        pthread_mutex_lock(&mutex); /* On verrouille le mutex */
        pthread_cond_wait (&condition, &mutex); /* On attend que la
condition soit remplie */
        printf("\nLE COMPTEUR A DÉPASSÉ 20.");
        pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
    }

    pthread_exit(NULL); /* Fin du thread */
}

```

Résultat :**Code : Console**

```

lucas@lucas-Desktop:~/Documents$ ./monprog

4
9
18
26
LE COMPTEUR A DÉPASSÉ 20.
9
18
23
LE COMPTEUR A DÉPASSÉ 20.
0
3
5
9
12

```

```
19
23
LE COMPTEUR A DÉPASSÉ 20.
0
8
9
10
17
25
LE COMPTEUR A DÉPASSÉ 20.
2
10
10
16
25
LE COMPTEUR A DÉPASSÉ 20.
8
10
18
26
LE COMPTEUR A DÉPASSÉ 20.
0
7
9
^C
```

Terminons ce chapitre par quelques moyen mnémotechniques qui peuvent vous permettre de retenir toutes les notions que l'on a apprises :

- Vous pouvez remarquer que toutes les variables et les fonctions sur les threads commencent par **pthread_**
- **pthread_create** : *create* = créer en anglais (donc créer le thread)
- **pthread_exit** : *exit* = sortir (donc sortir du thread)
- **pthread_join** : *join* = joindre (donc joindre le thread)
- **PTHREAD_MUTEX_INITIALIZER** : *initializer* = initialiser (donc initialiser le mutex)
- **pthread_mutex_lock** : *lock* = verrouiller (donc verrouiller le mutex)
- **pthread_mutex_unlock** : *unlock* = déverrouiller (donc déverrouiller le mutex)
- **pthread_cond_wait** : *wait* = attendre (donc attendre la condition)

(Comme quoi l'anglais ça sert des fois 🤪)

Les tubes

Dans ce cinquième chapitre, nous allons découvrir comment communiquer entre processus, et avec cela une nouvelle notion : les **tubes**.

Qu'est ce qu'un tube ?

Définition

Un **tube** (en anglais *pipe*) peut être représenté comme un tuyau (imaginaire, bien sûr ! 🤖) dans lequel circulent des informations.

Utilité

Les tubes servent à faire communiquer plusieurs processus entre eux.

Vocabulaire

On distingue alors les deux processus par leur action :

- Un écrit des informations dans le tube. Celui-ci est appelé **entrée** du tube ;
- L'autre lit les informations dans le tube. Il est nommé **sortie** du tube.



Le processus qui écrit ne peut pas lire des informations, et inversement. Il faut donc créer deux tubes si on veut que les processus établissent réellement un dialogue.

Manipuler les tubes

Créer

Pour commencer, il nous faudrait créer le tube. Pour cela, on utilise la fonction :

Code : C

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

Voici ses caractéristiques :

- Elle renvoie une valeur de type `int`, qui est de 0 si elle réussit, ou une autre valeur dans le cas contraire.
- Elle prend en argument un tableau de `int`, comprenant :
 - `descripteur[0]` : désigne la sortie du tube ;
 - `descripteur[1]` : désigne l'entrée du tube.

Mais là, nous devons faire face à notre premier problème. En effet, on ne crée le tube que dans un seul processus. L'autre ne peut donc pas en connaître l'entrée ou la sortie !

En conséquence, il faut utiliser `pipe` avant d'utiliser `fork`. Ensuite, le père et le fils auront les mêmes valeurs dans leur tableau `descripteur`, et pourront donc communiquer.

Écrire

Pour écrire dans un tube :

Code : C

```
ssize_t write(int entreeTube, const void *elementAEcrire, size_t
nombreOctetsAEcrire);
```

La fonction prend en paramètre l'**entrée du tube** (on lui enverra `descripteur[1]`), un **pointeur générique vers la mémoire contenant l'élément à écrire** (hummm, ça rappelle des souvenirs, non ? 😊) ainsi que le **nombre d'octets de cet élément**. Elle renvoie une valeur de type `ssize_t` correspondant au nombre d'octets effectivement écrits.

Exemple : Pour écrire le message "Bonjour" dans un tube, depuis le père :

Code : C

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

#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à
écrire */

int main(void)
{
    pid_t pid_fils;
    int descripteurTube[2];

    char messageEcrire[TAILLE_MESSAGE];

    pipe(descripteurTube);

    pid_fils = fork();

    if(pid_fils != 0) /* Processus père */
    {
        sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !");
        /* La fonction sprintf permet de remplir une chaîne de caractère
        avec un texte donné */
        write(descripteurTube[1], messageEcrire, TAILLE_MESSAGE);
    }

    return EXIT_SUCCESS;
}
```

Lire

Et pour lire dans un tube :

Code : C

```
ssize_t read(int sortieTube, void *elementALire, size_t
nombreOctetsALire);
```

La fonction prend en paramètre la **sortie du tube** (ce qui correspond à... `descripteur[0]`), un **pointeur vers la mémoire contenant l'élément à lire** et le **nombre d'octets de cet élément**. Elle renvoie une valeur de type `ssize_t` qui correspond au nombre d'octets effectivement lus. On pourra ainsi comparer le troisième paramètre (`nombreOctetsALire`) à la valeur renvoyée pour vérifier qu'il n'y a pas eu d'erreurs.

Exemple : Pour lire un message envoyé par le père à un fils :

Code : C

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

#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à lire */

int main(void)
{
    pid_t pid_fils;
    int descripteurTube[2];
    char messageLire[TAILLE_MESSAGE];

    pipe(descripteurTube);

    pid_fils = fork();

    if(pid_fils == 0) /* Processus fils */
    {
        read(descripteurTube[0], messageLire, TAILLE_MESSAGE);
        printf("Message reçu = \"%s\\n\"", messageLire);
    }

    return EXIT_SUCCESS;
}
```

Fermer

Lorsque nous utilisons un tube pour faire communiquer deux processus, il est important de fermer l'entrée du tube qui lit et la sortie du tube qui écrit.

En effet, il faut que le noyau voie qu'il n'y a plus de processus disposant d'un descripteur sur l'entrée du tube. Ainsi, dès qu'un processus tentera de lire à nouveau, il lui enverra EOF (fin de fichier).

Pour fermer une entrée ou une sortie :

Code : C

```
int close(int descripteur);
```

La valeur prise en paramètre correspond au descripteur de l'entrée ou de la sortie à fermer.

Exemple : Pour fermer l'entrée du processus fils et la sortie du processus père :

Code : C

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

int main(void)
{
    pid_t pid_fils;
    int descripteurTube[2];

    pipe(descripteurTube);
```

```

pid_fils = fork();

if(pid_fils == 0) /* Processus fils */
    close(descriptorTube[1]);

else /* Processus père */
    close(descriptorTube[0]);

return EXIT_SUCCESS;
}

```

Pratique

On passe à la pratique.

Écrivez un programme qui crée deux processus : le père écrit le message « Bonjour, fils. Je suis ton père ! ». Le fils le récupère, puis l'affiche.

Correction :

Code : C

```

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

#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à
lire et à écrire */

int main(void)
{
    pid_t pid_fils;
    int descriptorTube[2];

    unsigned char messageLire[TAILLE_MESSAGE],
    messageEcrire[TAILLE_MESSAGE];

    printf("Création du tube.\n");

    if(pipe(descriptorTube) != 0)
    {
        fprintf(stderr, "Erreur de création du tube.\n");
        return EXIT_FAILURE;
    }

    pid_fils = fork();

    if(pid_fils == -1)
    {
        fprintf(stderr, "Erreur de création du processus.\n");
        return 1;
    }

    if(pid_fils == 0)
    {
        printf("Fermeture de l'entrée dans le fils.\n\n");
        close(descriptorTube[1]);

        read(descriptorTube[0], messageLire, TAILLE_MESSAGE);
        printf("Nous sommes dans le fils (pid = %d).\nIl a reçu le
message suivant du père : \"%s\".\n\n\n", getpid(), messageLire);
    }

    else
    {
        printf("\nFermeture de la sortie dans le père.\n");
        close(descriptorTube[0]);
    }
}

```

```
    sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !");

    printf("Nous sommes dans le père (pid = %d).\nIl envoie le
message suivant au fils : \"%s\".\n\n\n", getpid(), messageEcrire);

    write(descripteurTube[1], messageEcrire, TAILLE_MESSAGE);

    wait(NULL);
}

return 0;
}
```

Résultat :

Code : Console

```
Création du tube.

Fermeture de la sortie dans le père.
Fermeture de l'entrée dans le fils.

Nous sommes dans le père (pid = 2584).
Il envoie le message suivant au fils : "Bonjour, fils. Je suis ton père !".

Nous sommes dans le fils (pid = 2585).
Il a reçu le message suivant du père : "Bonjour, fils. Je suis ton père !".
```

Notions complémentaires sur les tubes Entrées/Sorties et tubes

On peut lier la sortie d'un tube à `stdin` ou l'entrée d'un tube à `stdout`. Ensuite :

- Dans le premier cas : toutes les informations qui sortent du tube arrivent dans le flot d'entrée standard et peuvent être lues avec `scanf`, `fgets`, etc...
- Dans le second cas : toutes les informations qui sortent par `stdout` sont écrites dans le tube. On peut utiliser `printf`, `puts`, etc...

Pour faire cela, il suffit d'utiliser la fonction :

Code : C

```
int dup2(int ancienDescripteur, int nouveauDescripteur);
```

Le premier paramètre est facilement compréhensible, on lui enverra :

- `descripteur[0]` si on veut lier la sortie ;
- `descripteur[1]` si on veut lier l'entrée.

Le second paramètre correspond au nouveau descripteur que l'on veut lier au tube.
Il existe deux constantes, déclarées dans `unistd.h` :

- `STDIN_FILENO`;
- `STDOUT_FILENO`.

On utilise `STDIN_FILENO` lorsqu'on veut lier la sortie à `stdin`, et `STDOUT_FILENO` lorsqu'on veut lier l'entrée à `stdout`.

Exemple : Si on veut lier l'entrée d'un tube d'entrée et de sortie définies dans un tableau descripteur à `stdout` :

Code : C

```
dup2 (tube[1], STDOUT_FILENO);
```

Tubes nommés

Cependant, un problème se pose avec l'utilisation des tubes classiques. En effet, il faut obligatoirement que les processus connaissent le processus qui a créé le tube. Avec les tubes tout simples, il n'est pas possible de lancer des programmes indépendants, puis qu'ils établissent un dialogue.

C'est pourquoi on a créé une "extension" des tubes, appelée "**tube nommé**" (*named pipe*). Son concept est assez simple : comme son nom l'indique, le tube dispose d'un nom dans le système de fichier. Il suffit qu'un processus l'appelle par son nom, et hop, il accourt pour laisser le processus lire ou écrire en son intérieur. Sympa, hein ! 🤗

Créer

Pour commencer, il faut créer le tube. Concrètement, nous allons créer un fichier.

Pour créer un tube nommé, on utilise la fonction `mkfifo`, dont voici le prototype :

Code : C

```
int mkfifo (const char* nom, mode_t mode);
```

Le premier argument est le nom du tube nommé. On donne généralement l'extension « `.fifo` » au nom du tube nommé. Ne vous souciez pas pour l'instant de savoir à quoi correspond ce nom bizarre, vous le saurez en temps voulu.

Voici un exemple de nom pour un tube : « `essai.fifo` ».

Bon, jusque là pas de gros problèmes. En revanche, vous allez peut-être regretter d'avoir connu le deuxième paramètre... 🤗 En fait, il s'agit concrètement des droits d'accès du tube.



Euh... Et on est sensé trouver ça où ? 🤔

Deux solutions :

- La première, c'est de lire la [documentation du fichier `sys/stat.h`](#). Eh oui, c'est en anglais ! 🤗 Rendez-vous dans la section « File mode bits ». Vous y trouverez des constantes correspondant aux droits d'accès (`S_IRUSR`, `S_IWUSR`...). Vous pouvez combiner ces constantes avec le symbole « `|` ».
- Deuxième solution, pour les allergiques à l'anglais et pour ceux qui savent faire des additions : fabriquer des valeurs de droits tout seul, comme des grands ! 🤗

Les valeurs des droits comprennent quatre chiffres, sont sous mode octal : elles commencent donc par un zéro. Le premier chiffre correspond au propriétaire (vous en l'occurrence), le deuxième correspond au groupe du propriétaire

(Souvenir, souvenir! 🤔) et le troisième correspond à tous les autres. Une valeur est attribuée à chaque permission : 1 pour l'exécution, 2 pour l'écriture et 4 pour la lecture. On fait une somme quand on veut combiner plusieurs permissions. Bon, un petit exemple, parce que je sens que vous n'avez pas tout compris (😅) :

Pour attribuer toutes les permissions à vous, seule la lecture pour le groupe, et aucune pour les autres, la valeur correspondante est 0720 (Premier chiffre = 0 (obligatoire) ; Deuxième chiffre = 1 (Exécution) + 2 (Ecriture) + 4 (Lecture) ; Troisième chiffre = 0 (aucune permission)).

Ouf ! Tout ça pour un petit paramètre... 🤔

Enfin, dernière petite (🤔) chose : la fonction renvoie 0 si elle réussit, ou -1 en cas d'erreur. Vous pouvez aussi consulter la variable `errno`, qui peut contenir :

- **EACCES** : le programme n'a pas les droits suffisants pour accéder au chemin de création du tube nommé ;
- **EEXIST** : le tube nommé existe déjà ;
- **ENAMETOOLONG** : dépassement de la limitation en taille du nom de fichier (assez rare 🤔) ;
- **ENOENT** : le chemin du tube nommé n'existe pas ;
- **ENOSPC** : il n'y a plus assez de place sur le système de fichiers.

Comme vous pouvez le voir, il y a beaucoup de constantes possibles, donc généralement on ne les utilise pas dans un programme classique. Mais cela peut servir si votre programme ne marche pas très bien.

Bon, un petit exercice :

Créez un tube nommé que vous appellerez « *essai* », dont vous vous aurez toutes les permissions, dont le groupe aura les permissions de lecture et d'écriture et dont les autres n'auront aucun droits.

Deux solutions de possibles, pour chacune présentées pour les droits d'accès :

1)

Code : C

```
if (mkfifo("essai.fifo", S_IRWXU | S_IRGRP | S_IWGRP) == -1)
{
    fprintf(stderr, "Erreur de création du tube");
    exit(EXIT_FAILURE);
}
```

2)

Code : C

```
if (mkfifo("essai.fifo", 0760) == -1)
{
    fprintf(stderr, "Erreur de création du tube");
    exit(EXIT_FAILURE);
}
```

Ouvrir

Ensuite, il faut ouvrir l'entrée/la sortie du tube avec la fonction `open` :

Code : C

```
int open (const char* cheminFichier, int options);
```

La fonction renvoie une valeur de type `int` que l'on attribue à l'extrémité du tube en question.
Le premier argument est le nom du fichier (on mettra le nom du tube nommé, bien évidemment).
Le second argument indique si c'est l'entrée ou la sortie du tube. Il existe deux constantes pour cela, déclarées dans `fcntl.h` :

- `O_WRONLY` : pour l'entrée ;
- `O_RDONLY` : pour la sortie.

Exemple : Pour ouvrir l'entrée d'un tube « *essai.fifo* » :

Code : C

```
descripteur[1] = open("essai.fifo", O_WRONLY);
```

Ensuite, vous pouvez écrire et lire avec `write` et `read` comme si c'était des tubes classiques.

Bon, un dernier exercice avant de nous quitter pour ce chapitre :

Écrivez deux programmes indépendants : un écrit un message dans un tube nommé, et l'autre le lit, puis l'affiche. Exécutez ces deux programmes en même temps.

Correction :

Ecrivain.c :

Code : C

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

#define TAILLE_MESSAGE 256

int main(void)
{
    int entreeTube;
    char nomTube[] = "essai.fifo";

    char chaineAEcrire[TAILLE_MESSAGE] = "Bonjour";

    if(mkfifo(nomTube, 0644) != 0)
    {
        fprintf(stderr, "Impossible de créer le tube nommé.\n");
        exit(EXIT_FAILURE);
    }

    if((entreeTube = open(nomTube, O_WRONLY)) == -1)
    {
        fprintf(stderr, "Impossible d'ouvrir l'entrée du tube nommé.\n");
        exit(EXIT_FAILURE);
    }

    write(entreeTube, chaineAEcrire, TAILLE_MESSAGE);

    return EXIT_SUCCESS;
}
```


Lecteur.c :

Code : C

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

#define TAILLE_MESSAGE 256

int main(void)
{
    int sortieTube;
    char nomTube[] = "essai.fifo";

    char chaineALire[TAILLE_MESSAGE];

    if((sortieTube = open ("essai.fifo", O_RDONLY)) == -1)
    {
        fprintf(stderr, "Impossible d'ouvrir la sortie du tube nommé.\n");
        exit(EXIT_FAILURE);
    }

    read(sortieTube, chaineALire, TAILLE_MESSAGE);
    printf("%s", chaineALire);

    return EXIT_SUCCESS;
}
```

Résultat :

Premier terminal :

Code : Console

```
lucas@lucas-Desktop:~/Documents$ ./ecrivain
—
```

Deuxième terminal :

Code : Console

```
lucas@lucas-Desktop:~/Documents$ ./lecteur
Bonjour
```

Félicitations !

Vous venez de terminer le chapitre 5 !

Et bah dites donc ! On dirait que vous n'êtes plus des débutants en la matière maintenant : vous savez créer des processus et les faire communiquer, manipuler des *threads* et exécuter des programmes.

Il nous reste encore à maîtriser les signaux, les sémaphores, et bien d'autres choses encore avant de pouvoir passer dans une seconde partie plus difficile. Ça donne envie, hein ? 🤔



Ce tutoriel est en cours de construction.

Je suis ouvert à toute remarque ou critique constructive.

- N'hésitez pas à donner votre avis sur le sujet du forum consacré à ce tutoriel ([ici](#) ou [là](#)) ou dans ses commentaires.
- Faites moi part de vos suggestions par [MP](#) en débutant le titre de votre message par [PROGRAMMATION SYSTÈME UNIX] .
- Si il y a une notion que vous ne comprenez pas bien, posez votre question sur le [forum](#).