

Système d'exploitation Linux - Programmation avancée en C

Jean-Marc Wagner

10 septembre 2022

Avant-propos

Ce syllabus aborde les concepts de base du système d'exploitation Linux, ainsi que la programmation système en C sous Linux. Pour aborder ces notions, il est nécessaire de maîtriser la logique de programmation ainsi que les fondements du langage C.

La ligne conductrice de ce syllabus est le développement sous Linux d'applications comportant plusieurs processus s'exécutant simultanément et dès lors la programmation de la communication et de l'interaction entre ces processus.

Ce syllabus a été conçu sur base du cours de "Système d'exploitation UNIX" de Denys Mercenier, ancien professeur de la HEPL, pensionné en février 2021. Je le remercie ici pour son expérience et son aide au cours des années que j'ai déjà accomplies au sein du département "Sciences et techniques" de la HEPL.

Je remercie également mon collègue Patrick Quettier pour sa relecture attentive et les corrections apportées.

Bonne lecture !

Jean-Marc Wagner

Table des matières

1	Introduction	9
1.1	Généralités et bref historique	9
1.2	Distributions Linux	10
1.3	Généralités sur le développement sous Linux	11
1.4	Interaction avec le système : la ligne de commande	12
1.5	L'aide en ligne : la commande <code>man</code>	14
2	Compilation d'un programme	17
2.1	Le compilateur	17
2.1.1	Compilation d'un premier programme	17
2.1.2	L'option <code>-o</code>	18
2.1.3	Pré-compilation, compilation et édition de liens	19
2.1.4	Directives de pré-compilation	20
2.1.5	L'option <code>-D</code>	24
2.1.6	L'option <code>-I</code>	25
2.1.7	L'option <code>-Wall</code>	26
2.1.8	Problème d'édition de liens	27
2.2	Ecriture d'un premier programme	28
2.2.1	Tester les codes de retour - la variable globale <code>errno</code>	28
2.2.2	Convention d'écriture d'une fonction	32
2.2.3	Redirection des entrée/sorties standards	34
2.2.4	Ecriture d'un fichier de traces	38
2.2.5	Passage d'arguments à un programme : <code>argc</code> et <code>argv</code>	39
2.3	Ecriture d'une application en plusieurs fichiers	41
2.3.1	Création des fichiers <code>.cpp</code>	42
2.3.2	Création des fichiers <code>.h</code>	45
2.3.3	Le mot clé "extern"	47
2.4	Automatisation de la compilation : la commande <code>make</code>	49
2.4.1	Un premier <code>makefile</code>	50
2.4.2	Création de plusieurs exécutables avec le même fichier <code>makefile</code>	53
2.4.3	Paramétrisation d'un fichier <code>makefile</code>	54
3	Les fichiers sous Linux	61
3.1	Généralités	61
3.1.1	Utilisateurs et groupes	61
3.1.2	Les droits	62
3.1.3	Nom des fichiers	65
3.1.4	Les i-noeuds	66
3.1.5	Structure d'un disque	69
3.2	Appels bufferisés du système de fichiers	70

3.3	Appels système du système de fichiers	73
3.3.1	Tables gérées par le système	73
3.3.2	La primitive “open”	76
3.3.3	La primitive “close”	78
3.3.4	Les primitives “write” et “read”	80
3.3.5	La primitive “lseek”	82
3.3.6	La primitive “chmod”	85
3.3.7	La primitive “access”	86
3.3.8	La primitive “umask”	88
3.3.9	Les primitives “dup” et “dup2”	90
3.3.10	Les primitives “stat” et “fstat”	95
3.4	Appels bufferisés ou non bufferisés ?	99
3.4.1	Correspondance entre les fonctions de haut et de bas niveau	99
3.4.2	La fonction “fileno”	100
3.4.3	La fonction “fdopen”	103
4	Les processus	107
4.1	Introduction	107
4.1.1	Définition, structure et identifiant d'un processus	107
4.1.2	L'interpréteur de commandes	108
4.1.3	Lancement synchrone et asynchrone d'un processus	109
4.1.4	Table des processus et cycle de vie d'un processus	110
4.1.5	Identité et propriétaires d'un processus	113
4.2	Créer ses propres processus	117
4.2.1	Mécanisme de lancement d'une commande par le shell	117
4.2.2	Création d'un processus : l'appel système fork	119
4.2.3	Terminer un processus : l'appel système exit	127
4.2.4	Attendre la fin d'un processus : l'appel système wait	129
4.2.5	Recouvrement d'un processus : les fonctions exec()	138
4.2.6	Un exemple complet	144
5	Les signaux	147
5.1	Introduction	147
5.1.1	Généralités	147
5.1.2	Comportement par défaut vis à vis des signaux	149
5.1.3	Envoi d'un signal : la commande kill	150
5.1.4	Gestion et traitement des signaux reçus	152
5.2	Les ensembles de signaux : le type sigset_t	155
5.3	Armement d'un signal : la fonction sigaction	157
5.3.1	La structure sigaction	159
5.3.2	Prototype du handler de signal	160
5.3.3	Masquage des signaux pendant l'exécution du handler	162
5.3.4	Quelques variantes à l'armement standard	165
5.3.5	Application : Gestion des processus zombies, le signal SIGCHLD	170
5.4	Masquage de signaux : la fonction sigprocmask	174
5.4.1	Mise en place d'une section critique	176
5.4.2	Récupérer le vecteur des signaux pendants/reçus	178
5.5	Les sauts non locaux	181
5.5.1	Exemple : Traitement correct du signal SIGFPE	184
5.5.2	Mémorisation du masque de signaux dans le contexte	186

5.6	Les groupes de processus	188
5.6.1	Attacher un processus à un groupe	189
5.6.2	Envoi d'un signal à un groupe de processus	194
5.7	Quelques fonctions utiles	198
5.7.1	La fonction pause	198
5.7.2	La fonction sleep	199
5.7.3	Gestion d'un "Time Out" : la fonction alarm, le signal SIGALRM	201
6	Communication entre processus - Les IPC	205
6.1	Introduction	205
6.1.1	Généralités	205
6.1.2	Eléments communs à tous les IPC	205
6.2	Contrôle des IPC en ligne de commande	206
6.2.1	Visualiser les IPC : la commande ipcs	207
6.2.2	Supprimer les IPC : la commande ipcrm	209
6.3	Les files de messages	211
6.3.1	Structure interne d'une file de messages	211
6.3.2	Une espèce de "boîte aux lettres"	213
6.3.3	Type d'un message	214
6.3.4	Identifiant d'une file de messages	215
6.3.5	Obtention d'une file de messages : la fonction msgget	215
6.3.6	Suppression d'une file de messages : la fonction msgctl	219
6.3.7	Structure d'un message	222
6.3.8	Envoyer un message : la fonction msgsnd	223
6.3.9	Recevoir un message : la fonction msgrcv	226
6.3.10	Quelques variantes à la réception standard d'un message	230
6.3.11	Application : Client/Serveur mono-processus	232
6.4	Les mémoires partagées	237
6.4.1	Structure interne d'une mémoire partagée	238
6.4.2	Obtention d'une mémoire partagée : la fonction shmget	239
6.4.3	Suppression d'une mémoire partagée : la fonction shmctl	243
6.4.4	Attacher un processus à une mémoire partagée : la fonction shmat	247
6.4.5	Structurer une mémoire partagée	251
6.4.6	Détacher un processus d'une mémoire partagée : la fonction shmdt	253
6.5	Les sémaphores	256
6.5.1	Principe général	256
6.5.1.1	Un seule ressource partagée	257
6.5.1.2	Plusieurs ressources identiques partagées	258
6.5.1.3	Plusieurs ressources différentes partagées	259
6.5.1.4	Synchronisation pure	260
6.5.1.5	Premier aperçu des fonctions associées aux sémaphores	261
6.5.2	Structure interne d'un ensemble de sémaphores	261
6.5.3	Obtention d'un ensemble de sémaphores : la fonction semget	263
6.5.4	Initialisation/suppression d'un ensemble de sémaphores : semctl	268
6.5.5	Synchroniser les processus : la fonction semop	275
6.5.6	Fonctions simplifiées de synchronisation	280
6.5.7	Exemple : Accès synchronisé à une mémoire partagée	281

7 Les tubes de communications	287
7.1 Introduction	287
7.2 Mécanisme et caractéristiques d'un tube	287
7.3 Création d'un tube : la fonction pipe	288
7.4 Ecriture/Lecture dans un tube	291
7.4.1 La fonction write	291
7.4.2 La fonction read	293
7.5 Une communication Fils-Fils	296
7.5.1 Une lecture non bloquante	299
7.6 Redirection par tube de stdout vers stdin	301
7.6.1 Connexion par programmation	302
7.6.2 Connexion par ligne de commande	304
7.7 Une communication bidirectionnelle	305
8 Compléments sur les fichiers	307
8.1 Les verrous	307
8.1.1 Principe	307
8.1.2 La fonction fcntl	309
8.1.3 Pose non bloquante d'un verrou	312
8.1.4 Pose bloquante d'un verrou	315
8.1.5 Test de l'existence de verrous	317
8.2 Les répertoires	319
8.2.1 Crédation d'un répertoire	319
8.2.2 Suppression d'un répertoire	321
8.2.3 Changement de répertoire courant	322
8.2.4 Lecture du contenu d'un répertoire	325
8.2.5 Positionnement dans un fichier répertoire	328
9 Annexes	331
9.1 Quelques commandes utiles	331
9.1.1 La commande "lsof"	331
9.1.2 La commande "tar"	336
9.2 L'API C pour MySql	337
9.2.1 Une base de données exemple	337
9.2.2 Connexion/déconnexion à une base de données	339
9.2.3 Requêtes d'insertion, de modification et de suppression	342
9.2.4 Requête de sélection	345

Chapitre 1

Introduction

1.1 Généralités et bref historique

Linux est une famille de systèmes d'exploitation open source de type **Unix** fondé sur le **noyau Linux**, créé en 1991 par **LINUS TORVALDS**. A l'origine, Linux a été développé pour les ordinateurs compatibles PC mais il n'a jamais équipé qu'une très faible part des ordinateurs personnels. Les domaines principaux où les systèmes Linux sont prédominants sont

- les **services réseaux** : web, hébergement, cloud.
- les **systèmes embarqués** : industriels, scientifiques, IOT, ...
- les **mobiles** : le système d'exploitation Android est basé le noyau Linux.
- les **supercalculateurs**.

Le système Unix a été créé par la société AT&T au début des années 1970 et s'est notamment répandu dans le monde universitaire. En 1977, des chercheurs de l'université de Californie apportèrent de nombreuses améliorations au système Unix et le distribuèrent sous le nom de "Berkeley Software Distribution" (BSD). Finalement, les distributions BSD furent purgées du code AT&T et librement disponibles en 1991.

En 1983, **RICHARD STALLMAN**, un militant du logiciel libre, annonce son projet de développer un système d'exploitation libre compatible Unix appelé **GNU**. Dès 1985, certaines pièces maîtresses sont opérationnelles, dont le compilateur **GCC**. En 1991, l'étudiant finlandais **LINUS TORVALD**, indisposé par la faible disponibilité du serveur Unix de l'université d'Helsinki, entreprend le développement d'un noyau de système d'exploitation, qui prendra le nom de "**noyau Linux**". LINUS TORVALD choisit rapidement de publier son noyau sous licence GNU GPL. Cette décision rend compatibles juridiquement les systèmes **GNU** et **Linux**. **GNU** et le noyau **Linux** sont alors associés pour former un nouveau système d'exploitation appelé **GNU/Linux** ou plus simplement **Linux**.

A l'origine, l'installation d'un système d'exploitation opérationnel **GNU/Linux** nécessitait des connaissances solides en informatique et obligeait à trouver et installer les logiciels un à un. Rapidement, des ensembles de logiciels formant un système complet prêt à l'usage ont été disponibles : ce sont les premières distributions **Linux**.

1.2 Distributions Linux

On appelle **distribution GNU/Linux** (ou distribution Linux) une solution prête à être installée par l'utilisateur final comprenant

- le système d'exploitation GNU,
- le noyau Linux,
- des programmes d'installation et d'administration de l'ordinateur,
- un mécanisme facilitant l'installation et la mise à jour des logiciels comme RPM ou APT,
- une sélection de logiciels produits par d'autres développeurs.

Une distribution peut par exemple choisir de se spécialiser sur l'environnement de bureau **GNOME**, **KDE** ou **Xfce**. Elle est également responsable de la configuration par défaut du système (graphisme, simplicité, ...), du suivi de sécurité (installations de mises à jour) et plus généralement de l'intégration de l'ensemble.

La diversité des distributions permet de répondre à des besoins divers, qu'elles soient à but commercial ou non ; orientée serveur, bureautique ou embarqué ; orientée grand public ou public averti ; généraliste ou spécialisée pour un usage spécifique (pare-feu, routeur réseau, grappe de calcul, etc.) ; certifiées sur un matériel donné ; ou tout simplement entièrement libres, c'est-à-dire dépourvues de tout code propriétaire.

La plupart des distributions sont dérivées d'une autre distribution. Ainsi, trois distributions sont à l'origine de la plupart des autres :

1. **Slackware**, apparue en 1993, qui est aujourd'hui la plus ancienne distribution encore en activité, toujours maintenue par PATRICK VOLKERDING,
2. **Debian**, éditée par une communauté de développeurs,
3. **Red Hat**, éditée par l'entreprise américaine du même nom qui participe également au développement de **Fedora**.

De nombreuses autres distributions plus ou moins spécialisées existent, étant pour la plupart dérivées des projets précités. Par exemple voici quelques distributions spécialisées « environnement de bureau » :

- **Ubuntu**, éditée par Canonical Ltd. qui est dérivée de Debian,
- **MEPIS**, également fondée sur Debian,
- **Zenwalk**, dérivée de Slackware,
- **OpenMandriva Lx** et **Mageia** éditées par des associations à but non lucratif, dérivées de feu Mandriva Linux elle-même dérivée de Red Hat.

Il existe également beaucoup de distributions dites Live CD, l'une des plus célèbres est **Knoppix**. Elles offrent la possibilité de démarrer un système d'exploitation GNU/Linux complet

et d'accéder à de nombreux logiciels à partir du support (CD, DVD ou clé USB) sans installation préalable sur le disque dur, et sans altérer son contenu. Cette souplesse d'utilisation les rend très populaires pour les démonstrations d'utilisation de GNU/Linux, et sont même utilisées comme outils de maintenance système.

En ce qui nous concerne, nous utiliserons la distribution “**Oracle Linux 8**” basée elle-même sur la distribution de Red Hat “**Red Hat Enterprise Linux 8**” (**RHEL8**). Cette distribution fait suite à la disparition progressive du système d'exploitation Unix **Solaris** originellement développé par la société Sun Microsystems, société rachetée par la société Oracle en 2009.

1.3 Généralités sur le développement sous Linux

Dans une machine fonctionnant sous Linux, de nombreuses couches logicielles sont empilées, chacune fournissant des services aux autres. Il est important de comprendre comment fonctionne ce modèle pour savoir où une application viendra s'intégrer.

La base du système est le **noyau**, qui est le seul élément à porter véritablement le nom de “Linux”. Le noyau est souvent imaginé comme une sorte de logiciel mystérieux fonctionnant en arrière-plan pour surveiller les applications des utilisateurs, mais il s'agit avant tout d'un **ensemble cohérent de routines** (nous parlerons de “**processus**” dans la suite) fourni par le noyau et fournissant des **services** aux applications, en s'assurant de conserver l'intégrité du système. Pour le développeur, le noyau est surtout une **interface entre son application**, qui peut être exécutée par n'importe quel utilisateur, **et la machine physique** dont la manipulation directe doit être supervisée par un dispositif privilégié.

Le noyau fournit donc des **points d'entrée**, que l'on nomme “**appels système**” et que le programmeur appelle comme des sous-routines (ou encore “fonctions”) offrant des services variés. Par exemple, l'appel système **write()** permet d'écrire des données dans un fichier. L'application appelante n'a pas besoin de savoir sur quel type de système de fichiers, l'écriture se fera. Seul le noyau s'occupera de la basse besogne consistant à piloter les contrôleurs de disques, gérer la mémoire ou coordonner le fonctionnement des périphériques comme les cartes réseaux.

L'utilisation des appels systèmes est en principe suffisante pour écrire n'importe quelle application sous Linux. Toutefois, ce genre de développement serait particulièrement fastidieux, et la **portabilité** du logiciel résultant serait loin d'être assurée. Il existe donc une **couche supérieure** avec des fonctions qui viennent compléter les appels systèmes, permettant ainsi un développement plus facile et assurant également une meilleure portabilité des applications. Cette interface est constituée par la **bibliothèque C (libC)**.

Cette bibliothèque regroupe des fonctionnalités complémentaires de celles qui sont assurées par le noyau, par exemple tous les fonctions mathématiques (le noyau n'utilise pas les nombres réels). La bibliothèque C permet aussi d'**encapsuler les appels systèmes** dans des routines de plus haut niveau, qui sont donc plus aisément portables d'une machine à l'autre. Nous verrons par exemple que les descripteurs de fichiers fournis par l'interface du noyau restent limité à l'univers Unix, alors que les flux de données (par exemple la structure FILE utilisée par la fonction **fwrite()** de la librairie C) qui les encadrent sont portables sur tout système implémentant une bibliothèque C.

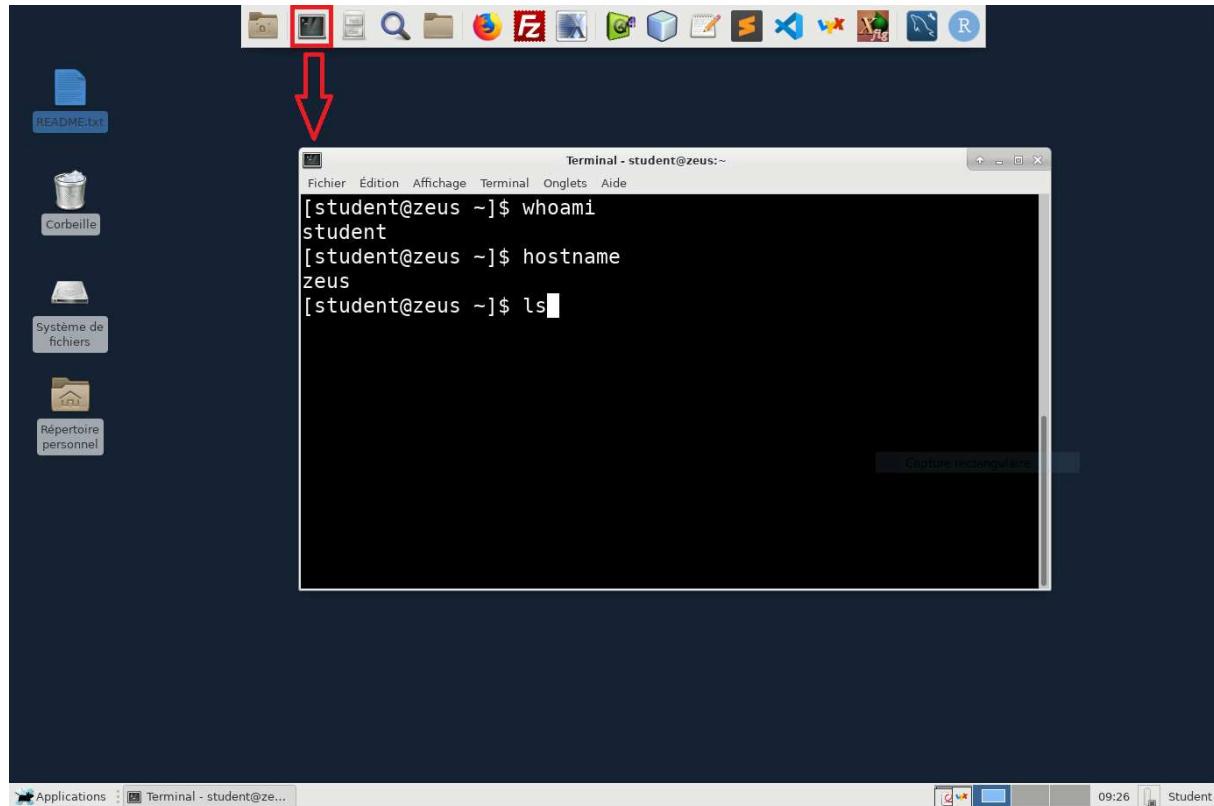
1.4 Interaction avec le système : la ligne de commande

De par la filiation avec UNIX, la **ligne de commande** (ou **shell Unix**) est toujours disponible dans GNU/Linux, quelle que soit la distribution.

Elle est restée longtemps incontournable, mais ce n'est plus vrai avec les distributions récentes et simples d'utilisation destinées à l'usage personnel, telles que Ubuntu ou Kubuntu. Néanmoins, les aides en ligne mentionnent souvent la marche à suivre en ligne de commande, même lorsqu'une configuration graphique est possible : cette méthode est **plus universelle dans le monde GNU/Linux**, et souvent plus facile à expliquer pour la personne qui aide, et son interlocuteur n'a qu'à copier-coller l'indication.

Une interface graphique bien conçue permet de nos jours d'accomplir la grande majorité des tâches bien plus agréablement, mais ce n'est pas toujours le cas, particulièrement lorsque la tâche a un aspect répétitif ou non prévu. La ligne de commande, qui tire sa puissance de sa **possibilité de combiner à l'infini des sous-tâches automatiques**, et qui permet presque naturellement d'automatiser la tâche ainsi accomplie, peut alors se révéler plus efficace que l'interface graphique. Scientifiques, ingénieurs et développeurs comptent parmi ses plus fréquents utilisateurs.

Interface graphique et ligne de commande peuvent aussi se compléter l'une et l'autre : KDE, GNOME et Xfce sont livrés avec un **terminal** pour piloter, et donc, automatiser toutes les applications graphiques depuis la ligne de commande. Voici un exemple de terminal dans l'environnement graphique Xfce de Oracle Linux 8 :



Pour la suite, il sera donc nécessaire de maîtriser un minimum les **commandes de base**, à savoir :

- **ls** : qui permet de lister les fichiers présents dans un répertoire.
- **cd** : qui permet de changer de répertoire.
- **cp** : qui permet de copier/coller un fichier.
- **mv** : qui permet de déplacer ou renommer un fichier.
- **mkdir** : qui permet de créer un répertoire.
- **rm** : qui permet de supprimer un fichier.
- **who** : qui permet d'afficher les utilisateurs connectés sur la machine.
- **ps** : qui permet d'afficher les processus en cours d'exécution sur la machine.
- **gcc/g++** : qui permet de compiler un programme.
- **hostname** : qui permet d'afficher le nom de la machine.
- **man** : qui fournit la documentation de toutes les commandes et fonctions du système.
- **tar** : qui permet de créer des archives contenant des fichiers et des sous-répertoires.
- ...

De plus, chacune de ces commandes possède un nombre impressionnant d'options. Le **format général** d'une commande est le suivant :

```
commande [-options...] [-argletter optarg] [cmdarg...]
```

où

- [] entourne une option ou un argument qui n'est pas exigé,
- ... indique un nombre variable d'options ou d'arguments,
- **commande** est le nom de la commande (qui est en fait un fichier exécutable),
- **options** est toujours précédé du caractère '-' , les différentes options peuvent être regroupées,
- **argletter** est toujours précédé du caractère '-' et désigne une option avec paramètre, **optarg** est son argument,
- **cmdarg** sont les arguments. Dans la majorité des cas, il s'agit de fichiers.

Exemple 1.1. Voici quelques exemples d'exécution de commandes :

```
# who
...
# ls -l
...
# g++ -o Prog Prog.cpp -lm -IMyLib
```

où nous constatons que

1. la commande **who** est utilisée sans option.
2. la commande **ls** est utilisée avec l'option **-l** (qui est une option sans paramètre) qui permet d'afficher, en plus du nom des fichiers, leur type, droits d'accès, etc...
3. L'option **-o** de la commande **g++** est une option avec paramètre, elle est suivie de ce paramètre qui est **Prog**.
4. **Prog.cpp** est l'**argument** de la commande **g++**, il s'agit du fichier que l'on veut compiler.
5. L'option **-lm** est en fait l'option **-l** qui est une option avec paramètre, ici **m**. Ceci spécifie que l'on veut faire l'édition de liens avec la librairie mathématique.
6. L'option **-I** de la commande **g++** est également une option avec paramètre, elle est suivie du paramètre **MyLib** qui est un répertoire.



Les paramètres de certaines commandes seront abordées en détails dans la suite en fonction des besoins.

1.5 L'aide en ligne : la commande man

Cette commande est l'une des plus importantes dans le monde Unix. Elle permet d'obtenir l'**aide en ligne** sur les **commandes**, les **fonctions de la bibliothèque C** et les **appels systèmes**. Elle peut paraître complexe mais surtout elle est complète et adaptée au système sur lequel vous travaillez.

L'aide en ligne (encore appelée le "man") est une "espèce d'encyclopédie", structurée en sections numérotées :

Section	Contenus
1	Commandes de l'interpréteur de commandes (shell).
2	Appels système (fonctions fournies par le noyau).
3	Appels de la bibliothèque C.
4	Fichiers spéciaux.
5	Formats des fichiers et conventions.
...	...

En ce qui nous concerne, nous serons particulièrement intéressés par les sections 1 à 3. La commande **man** prend en argument le nom de la commande, de l'appel système ou de la fonction dont on désire avoir le manuel. La commande man scrute alors les sections en partant de la première jusqu'à trouver la première entrée correspondante. Si l'on souhaite accéder directement à une section particulière, il faut utiliser l'option **-s** de la commande man.

Exemple 1.2. Voici quelques exemples d'exécution de la commande man :

```
# man exit
... // manuel de la commande exit
# man -s 2 exit
... // manuel de l'appel système exit
```

où nous constatons que

1. **exit** est en fait une commande et un appel système qui portent le **même nom**.
2. le **premier appel** de la commande **man** a fourni le manuel de la **commande exit**. En effet, celle-ci se trouve dans la section 1 du manuel et vu que la recherche commence par défaut du début, c'est elle qui est trouvée en premier.
3. Le **second appel** de la commande **man** a fourni le manuel de l'**appel système exit**. En effet, la section 2 a été précisée avec l'option **-s** et la recherche a été réalisée dans la section (2) des appels système.



Le manuel affiché pour une fonction, un appel système ou une commande est structuré selon plusieurs sous-sections :

Sous-Section	Contenus
Nom	Nom et brève description de la commande ou de la fonction.
Synopsis	Syntaxe de la commande ou de la fonction avec les libraires éventuelles à inclure (les fichier headers d'extension .h)
Description	Description complète.
Valeurs de retour	Valeurs de retour possibles. Ceci sera particulièrement important lorsque nous testerons si une fonction a correctement réalisé la tâche demandée.
Erreurs	Liste des différentes erreurs possibles.
Voir aussi	Renvoit vers d'autres sections du manuel en rapport avec la commande ou la fonction recherchée.
...	...

Exemple 1.3. Voici un exemple d'utilisation de la commande man permettant d'obtenir l'aide en ligne de la fonction printf de la bibliothèque C :

```
# man -s 2 printf
PRINTF(3)          Manuel du programmeur Linux          PRINTF(3)

NOM
    printf, fprintf, sprintf, snprintf, ...
    - Formatage des sorties

SYNOPSIS

#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
...

DESCRIPTION

    Les fonctions de la famille printf() produisent des sorties en accord avec le
    format décrit plus bas. Les fonctions printf() et vprintf() écrivent leur sortie
    sur stdout, le flux de sortie standard. fprintf() et vfprintf() écrivent sur le
    flux stream indiqué. sprintf(), snprintf(), vsprintf() et vsnprintf() écrivent leurs
    sorties dans la chaîne de caractères str.
...
```

où nous constatons que

1. le **titre** nous indique que nous sommes bien dans la **section 3** et qu'il s'agit bien de la fonction **printf**.
2. La documentation ne concerne pas que printf mais **toutes les fonctions de la même famille** dont il ne faut pas oublier d'inclure le fichier **stdio.h** dans le programme qui les utilise.
3. Pour **sortir** du man, il faut taper la touche '**q**'.



Chapitre 2

Compilation d'un programme

2.1 Le compilateur

Pour compiler, le système propose essentiellement les compilateurs suivants, utilisables sous forme de commande :

1. **gcc** pour la compilation de programmes écrits en C,
2. **g++** pour la compilation de programme écrit en C++.

Néanmoins, les différences et points communs entre gcc et g++ sont assez subtiles et nous n'entrerons pas dans les détails ici. Pour la suite, nous utiliserons exclusivement la commande **g++**.

Exemple 2.1. Le compilateur g++ s'utilise en ligne de commande :

```
# g++
g++: fatal error: no input files
compilation terminated.

# g++ --version
g++ (GCC) 6.3.1 20170216 (Red Hat 6.3.1-3)
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
```

L'option **--version** a permis d'afficher la version du compilateur. Ce n'est pas la dernière mais c'est celle qui est utilisée dans tous les exemples qui suivront.

2.1.1 Compilation d'un premier programme

Exemple 2.2. Soit le fichier source suivant (obtenu à l'aide de n'importe quel éditeur) :

```
***** HelloWorld.cpp ****/

int main()
{
    printf("Hello Linux World !\n");
    return 0;
}
```

Dans un terminal, nous pouvons compiler ce programme :

```
# cat HelloWorld.cpp
/***** HelloWorld.cpp *****/
int main()
{
    printf("Hello Linux World !\n");
    return 0;
}
# g++ HelloWorld.cpp
HelloWorld.cpp: In function ‘int main()’:
HelloWorld.cpp:5:32: error: ‘printf’ was not declared in this scope
    printf("Hello Linux World !\n");
                           ^
#
#
```

Dans cet exemple, une erreur de compilation nous indique que la fonction printf n'est pas connue et la compilation n'a pas abouti. Il manque bien évidemment le bon #include.



Exemple 2.3. Soit le fichier source suivant :

```
/***** HelloWorld.cpp *****/
#include <stdio.h>

int main()
{
    printf("Hello Linux World !\n");
    return 0;
}
```

Une nouvelle compilation fournit :

```
# g++ HelloWorld.cpp
# ls -l
total 16
-rwxrwxr-x. 1 student student 8496 21 fév 10:53 a.out
-rw-r--r--. 1 student student 109 21 fév 10:53 HelloWorld.cpp
# ./a.out
Hello Linux World!
#
```

La compilation a abouti sans erreur et le résultat est un fichier appelé **a.out**. Il s'agit de notre fichier exécutable que l'on peut lancer comme s'il s'agissait d'une commande.



2.1.2 L'option **-o**

Le nom du fichier exécutable obtenu par la compilation est a.out mais il est possible de donner un **nom différent au fichier de sortie** en utilisant l'option **-o** suivie du nom souhaité.

Exemple 2.4. La compilation du programme de l'exemple 2.3 peut également être réalisée par :

```
# ls -l
total 4
-rw-r--r--. 1 student student 109 21 fév 10:53 HelloWorld.cpp
# g++ HelloWorld.cpp -o HelloWorld
# ls -l
total 16
-rwxrwxr-x. 1 student student 8496 22 fév 16:58 HelloWorld
-rw-r--r--. 1 student student 109 21 fév 10:53 HelloWorld.cpp
# ./HelloWorld
Hello Linux World!
# rm HelloWorld
# g++ -o HelloWorld HelloWorld.cpp
# ls -l
total 16
-rwxrwxr-x. 1 student student 8496 22 fév 16:58 HelloWorld
-rw-r--r--. 1 student student 109 21 fév 10:53 HelloWorld.cpp
# ./HelloWorld
Hello Linux World!
#
```

Le fichier source et l'option -o peuvent être permuts sur la ligne de compilation mais le nom du fichier de sortie doit obligatoirement suivre le -o.

2.1.3 Pré-compilation, compilation et édition de liens

En réalité, la commande g++ (ou gcc) fait bien plus que compiler le fichier source. Elle réalise les opérations suivantes :

1. La **pré-compilation** réalisée par le pré-compilateur ou le pré-processeur (commande nommée **cpp**), qui gère toutes les **directives** #define, #include, #ifdef, #ifndef, ... du fichier source.
2. La **compilation** proprement dite qui transforme le code source prétraité en fichier contenant le code assembleur (fichier d'extension **.s**).
3. L'**assemblage** (commande nommée **as**) qui fournit des **fichiers "objets"** (d'extension **.o**) qui contiennent le code binaire avec des références aux fonctions et variables externes (contenues dans des librairies ou autres fichiers objets).
4. L'**édition de liens** (commande nommée **ld**) qui assure le regroupement des fichiers objets et des librairies pour fournir enfin le fichier exécutable.

Il est possible d'arrêter la compilation après une de ces étapes. Pour cela, il faut utiliser les options suivantes de g++ (ou gcc) :

Option	But
-E	Arrêter après la pré-compilation, avant la compilation
-S	Arrêter après la compilation, avant l'assemblage
-c	Arrêter après l'assemblage, laissant les fichiers objets disponibles

Nous allons utiliser l'option -E de g++ afin d'étudier quelques directives de pré-compilation.

2.1.4 Directives de pré-compilation

Les directives de pré-compilation font essentiellement un travail de “traitement de textes”, du style “copier/coller/remplacer” avec ou sans condition.

La directive #include

Cette directive ouvre le fichier texte qui suit l’ #include et remplace celui-ci par le contenu de ce fichier texte.

Exemple 2.5 (#include). Soit le fichier “MaStructure.h” suivant :

```
struct Data
{
    int valeur;
};
```

et le programme “Test_include.cpp” suivant :

```
#include "MaStructure.h"

int main()
{
    struct Data a;
    a.valeur = 5;

    return 0;
}
```

Arrêtons le processus de compilation après la pré-compilation :

```
# g++ Test_include.cpp -E
# 1 "Test_include.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Test_include.cpp"
# 1 "MaStructure.h" 1
struct Data
{
    int valeur;
};
# 2 "Test_include.cpp" 2

int main()
{
    struct Data a;
    a.valeur = 5;
    return 0;
}
```

Le pré-compilateur a écrit le résultat de son travail directement sur sa sortie standard (stdout). On voit clairement que l’ #include “MaStructure.h” a été remplacé par le contenu du

fichier.



Il est important de noter que

1. `#include "XXX"` : recherche le fichier XXX dans le **répertoire courant**. S'il ne le trouve pas, il y aura erreur. Le plus souvent, il s'agit de fichiers de l'utilisateur.
2. `#include <XXX>` : recherche le fichier XXX dans les **librairies installées dans le système**. Le compilateur sait où il doit aller pour retrouver le fichier. Par exemple, lors d'un `#include <stdio.h>`, le système a été configuré lors de l'installation pour que le compilateur trouve automatiquement le fichier stdio.h.

La directive `#define`

Cette directive permet de définir

- une **macro** et de lui attribuer une **valeur**. Lors de la pré-compilation, chaque occurrence de cette macro est remplacée par sa valeur. Il ne s'agit pas d'une variable du programme, aucune zone mémoire n'est réservée pour elle puisqu'il s'agit simplement d'une opération de remplacement dans le texte.
- une **macro sans valeur**. Une macro peut simplement être définie ou non. Selon qu'elle l'est ou pas, cela permet de faire des tests logiques. Nous y reviendrons plus loin.
- une **macro paramétrée** et de lui attribuer une expression dépendant de paramètres. A nouveau, lors de la pré-compilation, chaque occurrence de cette macro est remplacée par la valeur de son expression.

Le **nom d'une macro** peut être choisie arbitrairement par l'utilisateur. Néanmoins, les conventions recommandent d'utiliser des **majuscules** afin de ne pas les confondre avec les variables.

Exemple 2.6 (#define). Soit le programme suivant :

```
***** Test_define1.cpp *****
// Auteur : J-M Wagner
#define N 100

int main()
{
    int v[N];
    v[3] = 5 + N;

    return 0;
}
```

Arrêtons le processus de compilation après la pré-compilation :

```
# g++ Test_define1.cpp -E
# 1 "Test_define1.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Test_define1.cpp"

int main()
{
    int v[100];
    v[3] = 5 + 100;
    return 0;
}
```

Le pré-compilateur a remplacé chaque occurrence de la macro N par la valeur numérique 100. Contrairement à v, N n'est pas une variable du programme. Remarquez que les commentaires ont disparu lors de la pré-compilation.



Exemple 2.7 (macro paramétrée). Soit le programme suivant :

```
***** Test_define2.cpp *****/
#define MON_OPERATION(a,b) (a+b)*3

int main()
{
    int x=2, y=3, z;
    z = MON_OPERATION(x,y);

    return 0;
}
```

Arrêtons le processus de compilation après la pré-compilation :

```
# g++ Test_define2.cpp -E
# 1 "Test_define2.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Test_define2.cpp"

int main()
{
    int x=2, y=3, z;
    z = (x+y)*3;

    return 0;
}
```

Il faut noter que MON_OPERATION(a,b) n'est **pas une fonction** ! Lors de l'exécution, il n'y aura pas d'appel de fonction vu qu'il n'y a pas de fonction dans le fichier de sortie du pré-compilateur.



Les directives #ifdef, #ifndef et #endif

Ces directives permettent d'inclure ou d'exclure des lignes de code selon qu'une macro est définie ou non. La syntaxe générale est la suivante :

```
#ifdef macro
...
...
#endif
```

Si la macro est définie, les lignes de code situées entre #ifdef et #endif sont traitées par le pré-compilateur. Sinon, elles sont ignorées. Dans le cas du #ifndef, le principe est le même sauf que la macro ne doit pas être définie pour que les lignes de code soit traitées.

Exemple 2.8 (#ifdef). Soit le programme suivant :

```
***** Test_ifdef.cpp *****/
#define DEBUG

int main()
{
#ifdef DEBUG
    printf("Declaration des variables\n");
#endif
    int x=2, y=3, z;
#ifdef DEBUG
    printf("Calcul de z\n");
#endif
    z = x+y;
    return 0;
}
```

Arrêtons le processus de compilation après la pré-compilation :

```
# g++ Test_ifdef.cpp -E
# 1 "Test_ifdef.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Test_ifdef.cpp"

int main()
{
    printf("Declaration des variables\n");
    int x=2, y=3, z;
    printf("Calcul de z\n");
    z = x+y;
    return 0;
}
```

Si la ligne `#define DEBUG` est par contre mise en commentaire, cela donne :

```
# g++ Test_ifdef.cpp -E
# 1 "Test_ifdef.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Test_ifdef.cpp"

int main()
{
    int x=2, y=3, z;
    z = x+y;
    return 0;
}
#
```

Selon que la macro DEBUG (pour rappel, le nom d'une macro peut être choisi de manière arbitraire) est définie ou non, on peut donc imaginer d'inclure ou non du code de debuggage ou de traçage, et cela sans devoir remodifier tout le code source.



2.1.5 L'option -D

Afin d'utiliser les directives `#ifdef` ou `#ifndef`, il est nécessaire qu'une macro soit définie ou non. A l'**exemple 2.8**, nous avons dû avoir la ligne

```
#define DEBUG
```

dans notre programme afin de définir la macro DEBUG ou la mettre en commentaire selon nos besoins. Dans les deux cas, il a fallu modifier le code source, ce qui n'est pas pratique.

L'option **-D** du compilateur `g++` (ou `gcc`) permet de résoudre ce problème. Elle permet en **ligne de commande** de **définir une macro** passée en paramètre. Le nom de la macro doit obligatoirement suivre le `-D`.

Exemple 2.9 (l'option -D). Soit le programme suivant :

```
***** Test_ifdef2.cpp *****/
#include <stdio.h>

int main()
{
#ifndef DEBUG
    printf("Déclaration des variables\n");
#endif
    int x=2, y=3, z;
#ifndef DEBUG
    printf("Calcul de z\n");
#endif
    z = x+y;
```

```
printf("z = %d + %d = %d\n",x,y,z);
return 0;
}
```

Nous pouvons le compiler avec ou sans l'option -D, ce qui donne :

```
# g++ Test_ifdef2.cpp -D DEBUG
# ./a.out
Declaration des variables
Calcul de z
z = 2 + 3 = 5
# g++ Test_ifdef2.cpp
# ./a.out
z = 2 + 3 = 5
#
```

L'option -D DEBUG agit exactement comme si l'on avait fait un #define DEBUG dans le code source du programme.



2.1.6 L'option -I

L'option **-I** du compilateur g++ (ou gcc) est directement liée à la directive **#include**. Supposons qu'un utilisateur inclut dans son programme un fichier XXX.h à l'aide de la ligne

```
#include "XXX.h"
```

Le fichier XXX.h doit absolument se trouver dans le **répertoire courant**. Afin de classer ses fichiers, si le programmeur décide de déplacer ce fichier dans un répertoire, disons RRR, il devra alors changer son code source :

```
#include "RRR/XXX.h"
```

Dès lors, à chaque déplacement du fichier, le code source devra être modifié, ce qui n'est pas pratique.

L'option **-I** permet de résoudre ce problème. L'option **-I suivie du nom d'un répertoire** spécifie au compilateur que si une directive **#include** cherche un fichier et que celui-ci n'est pas présent dans le répertoire courant, il peut aller chercher dans le répertoire précisé. Le nom du répertoire doit obligatoirement suivre le -I.

Exemple 2.10 (l'option -I). Reprenons l'[exemple 2.5](#) et déplaçons le fichier MaStructure.h dans le répertoire MaLib :

```
# mkdir MaLib
# mv MaStructure.h MaLib/
# ls MaStructure.h
ls: impossible d'accéder à MaStructure.h: Aucun fichier ou dossier de ce type
# ls MaLib/
MaStructure.h
# g++ Test_include.cpp
Test_include.cpp:1:25: fatal error: MaStructure.h: Aucun fichier ou dossier de ce
type
 #include "MaStructure.h"
 ^
compilation terminated.
# g++ Test_include.cpp -I MaLib
# ./a.out
#
```

L'option -I MaLib indique au compilateur que si un #include cherche un fichier (qui ne se trouve pas dans le répertoire courant), il peut le chercher dans le répertoire MaLib.



2.1.7 L'option -Wall

Lors de la compilation, il se peut que certains avertissements ("warnings") de moindre gravité soient omis par le compilateur. Il est possible d'afficher tous les warnings grâce à l'option **-Wall**.

Exemple 2.11 (l'option -Wall). Soit le programme suivant :

```
***** Test_Wall.cpp *****
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    printf("taille int : %d\n", sizeof(int));
    printf("taille size_t : %d\n", sizeof(size_t));
    exit(0);
}
```

Nous pouvons le compiler avec ou sans l'option -Wall, ce qui donne :

```
# g++ Test_Wall.cpp
# g++ Test_Wall.cpp -Wall
Test_Wall.cpp: In function ‘int main()’:
Test_Wall.cpp:8:44: warning: format ‘%d’ expects argument of type ‘int’, but
argument 2 has type ‘long unsigned int’ [-Wformat=]
    printf("taille int : %d\n",sizeof(int));
                                         ^
Test_Wall.cpp:9:47: warning: format ‘%d’ expects argument of type ‘int’, but
argument 2 has type ‘long unsigned int’ [-Wformat=]
    printf("taille size_t : %d\n",sizeof(size_t));
                                         ^
#

```

Lors d'un printf, %d permet d'afficher un **int**. Or le retour de **sizeof** n'est pas de type int mais **size_t** (type défini dans le fichier stddef.h et équivalent à **unsigned long**), ce qui ne pose pas un réel problème lors de l'affichage mais il y a tout de même un conflit de type que l'option **-Wall** permet de mettre en évidence. Pour éviter ce warning, il faut remplacer le **%d** par un **%ld**.



Exemple 2.12 (l'option -Wall). Soit le programme suivant :

```
***** Test_Wall2.cpp *****
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;
    printf("Premier programme\n");
    exit(0);
}
```

Nous pouvons le compiler avec ou sans l'option -Wall, ce qui donne :

```
# g++ Test_Wall2.cpp
# g++ Test_Wall2.cpp -Wall
Test_Wall2.cpp: In function ‘int main()’:
Test_Wall2.cpp:7:7: warning: unused variable ‘i’ [-Wunused-variable]
    int i;
          ^
#

```

Dans ce cas-ci, l'option **-Wall** permet d'afficher les warnings concernant les variables déclarées mais non utilisées.



2.1.8 Problème d'édition de liens

Une erreur de compilation courante est celle qui apparaît lorsqu'une **fonction** est **déclarée** mais **non définie**. On parle alors de problème d'édition de liens.

Exemple 2.13 (Problème d'édition de liens). Soit le programme suivant :

```
***** ProblemeEditionLiens.cpp *****
#include <stdio.h>
#include <stdlib.h>

void Affiche(const char*);

int main()
{
    Affiche("Dans programme\n");
    exit(0);
}
```

Selon que nous compilons avec ou sans l'option **-c**, nous obtenons :

```
# g++ ProblemeEditionLiens.cpp -c
# g++ ProblemeEditionLiens.cpp
/tmp/ccgvCuSv.o : Dans la fonction « main » :
ProblemeEditionLiens.cpp:(.text+0xa) : référence indéfinie vers « Affiche(char
const*) »
collect2: error: ld returned 1 exit status
#
```

L'option **-c** arrête la compilation juste avant l'édition de liens. On observe donc bien que la compilation se réalise sans problème. Par contre, sans l'option **-c**, une **erreur d'édition de liens** se produit. En effet, la commande **ld** s'est arrêtée avec une erreur. Celle-ci recherche le code compilé de la fonction **Affiche()** (qui aurait été obtenu à partir de sa définition - manquante ici) et ne le trouve pas. Le code compilé de la fonction **Affiche()** aurait été **"lié"** au code compilé de la fonction **main()**, ainsi qu'au code compilé des bibliothèques utilisées, afin de fournir le fichier exécutable.



2.2 Ecriture d'un premier programme

2.2.1 Tester les codes de retour - la variable globale **errno**

Lors d'un appel de fonction, il se peut que tout ne se passe pas correctement : mauvais paramètre passé à la fonction, plus assez de mémoire, plus d'espace disque, ... Dans ce cas, une **erreur d'exécution** est détectée et celle-ci doit être traitée. Pour rappel, le prototype d'une fonction est le suivant :

```
typeDeRetour nomFonction(type1 paramètre1, type2 paramètre2, ...);
```

Sous Linux, le **type de retour** d'une fonction est généralement un **entier** et c'est grâce à cet entier que l'on sait si une erreur s'est produite lors de l'exécution de la fonction. De plus, lors de la compilation d'un programme, une **variable globale entière** (dite "implicite" - vous n'avez pas besoin de la déclarer pour qu'elle existe) appelée **errno** est automatiquement déclarée et sert à contenir un **code d'erreur spécifique** à l'erreur qui s'est produite. A chaque valeur (int) de **errno** correspond un message d'erreur, défini dans le fichier **errno.h**.

D'une manière générale, lors d'un appel de fonction (dont le type de retour est un **int**), on doit tester sa valeur de retour :

1. Si celle-ci est **positive ou nulle**, la fonction s'est déroulée **correctement** et le programme peut continuer son exécution.
2. Si celle-ci est **négative**, une **erreur** s'est produite lors de l'exécution de la fonction et la variable globale **errno** a été positionnée à une **valeur positive**. Grâce à la valeur de **errno**, il est alors possible de connaître la cause de l'erreur et de réagir en fonction :
 - (a) **soit arrêter** le programme : l'appel échoué de la fonction utilise une ressource non disponible et donc le programme n'a plus de raison d'être.
 - (b) **soit agir** pour remédier au problème rencontré : soit relancer l'appel de la fonction qui a été interrompue (voir le chapitre sur les signaux), soit relancer l'appel de la fonction avec d'autres paramètres ou appeler une autre fonction.

Quelques remarques :

1. La variable globale **errno** est modifiée lorsqu'une erreur se produit. Elle n'est **pas remise à 0 automatiquement** lors d'un appel de fonction qui se déroule correctement. Il est donc nécessaire de remettre **errno** à 0 manuellement si on souhaite continuer l'exécution du programme.
2. Certaines fonctions du langage C ne retournent pas un int. Par exemple, l'appel de **malloc(...)** retourne un pointeur. Si l'allocation dynamiquement de mémoire ne s'est pas déroulée correctement, la valeur de retour est NULL. Il suffit ensuite de consulter la variable **errno** pour connaître la cause de l'erreur. Il est donc nécessaire de consulter le **man** d'une fonction utilisée afin d'en connaître ses valeurs de retour.

En pratique, à chaque valeur entière de la variable **errno** correspond une **macro** qui est définie dans errno.h (ou un des fichiers inclus par errno.h), ainsi que le message d'erreur correspondant :

```
# cd /usr/include/asm-generic/
# more errno-base.h
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM      1 /* Operation not permitted */
#define ENOENT     2 /* No such file or directory */
#define ESRCH      3 /* No such process */
#define EINTR      4 /* Interrupted system call */
#define EIO         5 /* I/O error */
#define ENXIO      6 /* No such device or address */
#define E2BIG       7 /* Argument list too long */
#define ENOEXEC    8 /* Exec format error */
#define EBADF      9 /* Bad file number */
#define ECHILD    10 /* No child processes */
#define EAGAIN    11 /* Try again */
#define ENOMEM    12 /* Out of memory */
#define EACCES    13 /* Permission denied */
#defineEFAULT   14 /* Bad address */
#define ENOTBLK   15 /* Block device required */
#define EBUSY     16 /* Device or resource busy */
#define EEXIST    17 /* File exists */
#define EXDEV     18 /* Cross-device link */
#define ENODEV    19 /* No such device */
#define ENOTDIR   20 /* Not a directory */
#define EISDIR    21 /* Is a directory */
#define EINVAL    22 /* Invalid argument */
#define ENFILE     23 /* File table overflow */
#define EMFILE    24 /* Too many open files */
#define ENOTTY    25 /* Not a typewriter */
#define ETXTBSY   26 /* Text file busy */
#define EFBIG     27 /* File too large */
#define ENOSPC    28 /* No space left on device */
#define ESPIPE    29 /* Illegal seek */
#define EROFS     30 /* Read-only file system */
#define EMLINK    31 /* Too many links */
#define EPIPE     32 /* Broken pipe */
#define EDOM      33 /* Math argument out of domain of func */
#define ERANGE    34 /* Math result not representable */

#endif
#
```

Les valeurs possibles de errno ne s'arrêtent pas à 34 et d'autres macros sont définies dans d'autres fichiers inclus par errno.h.

Il existe plusieurs fonctions permettant d'aller lire facilement le fichier errno.h afin d'obtenir le message associé à une valeur particulière de errno. La première est la suivante :

```
#include <string.h>  
  
char *strerror(int errnum);
```

Cette fonction retourne une chaîne de caractères contenant le message d'erreur associé à la valeur errnum de errno.

Exemple 2.14 (strerror). Soit le programme suivant :

```
***** Test_strerror.cpp *****/  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
int main()  
{  
    for (int i=0 ; i<20 ; i++)  
        printf("errno = %2d --> %s\n",i,strerror(i));  
  
    exit(0);  
}
```

dont l'exécution fournit :

```
# ./Test_strerror  
errno = 0 --> Success  
errno = 1 --> Operation not permitted  
errno = 2 --> No such file or directory  
errno = 3 --> No such process  
errno = 4 --> Interrupted system call  
errno = 5 --> Input/output error  
errno = 6 --> No such device or address  
errno = 7 --> Argument list too long  
errno = 8 --> Exec format error  
errno = 9 --> Bad file descriptor  
errno = 10 --> No child processes  
errno = 11 --> Resource temporarily unavailable  
errno = 12 --> Cannot allocate memory  
errno = 13 --> Permission denied  
errno = 14 --> Bad address  
errno = 15 --> Block device required  
errno = 16 --> Device or resource busy  
errno = 17 --> File exists  
errno = 18 --> Invalid cross-device link  
errno = 19 --> No such device  
#
```

Remarquons que la valeur 0 de errno correspond à "Success", qui n'est pas un cas d'erreur.



Une autre manière de faire pour afficher le message d'erreur associé à la valeur courante de errno est d'utiliser la fonction suivante :

```
#include <stdio.h>

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

Cette fonction affiche sur la sortie d'erreur (stderr) du processus la chaîne de caractères **s** concaténée avec le caractère : et le message d'erreur associé à la **valeur courante** de errno.

Exemple 2.15 (perror). Soit le programme suivant :

```
***** Test_perror.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    errno = 2;
    perror("Erreur detectee");

    errno = 0;
    perror("Errno remise a zero");

    exit(0);
}
```

dont l'exécution fournit :

```
# ./Test_perror
Erreur detectee: No such file or directory
Errno remise a zero: Success
#
```

La manipulation de la variable **errno** nécessite l'inclusion du fichier **errno.h**. ■

2.2.2 Convention d'écriture d'une fonction

Lorsqu'un programmeur doit écrire ses propres fonctions, il est souhaitable que celles-ci respectent les mêmes conventions que celles pré définies dans le système. En particulier,

- Toujours (dans la mesure du possible), écrire une fonction de **type int**.
- Eviter une liste trop longue de **paramètres** : 2 ou 3 mais pas beaucoup plus. Un nombre trop grand de paramètres alourdit considérablement la programmation. Si cela est nécessaire, il est alors préférable d'utiliser une structure en paramètre.
- Veiller à déclarer les **variables en local** dans la fonction, cela pour éviter les conflits avec d'autres fonctions et éviter les problèmes de "réentrance" (voir plus tard).
- Ne pas afficher quoi que ce soit dans la fonction (sauf si c'est une fonction dédiée à cela...). En particulier, on **n'affiche pas la cause d'une erreur** dans la fonction. En effet, afficher un message en cas d'erreur est déjà une réaction à cette erreur. On pourrait choisir de réagir autrement en cas d'erreur. Donc la convention est de retourner un code d'erreur propre à

la fonction (par convention et généralement une valeur négative ou un pointeur NULL). Le traitement de l'erreur (comme par exemple l'affichage de la cause de l'erreur) se réalise dans le main() ou dans la fonction appelante.

Exemple 2.16 (Convention d'écriture d'une fonction). Soit le programme suivant :

```
***** ExempleFonction.cpp *****/
#include <stdio.h>
#include <stdlib.h>

FILE *OuvertureFichier(const char *,int);

int main()
{
    FILE *fb;

    if ((fb = OuvertureFichier("Fich.dat",1000)) == NULL)
    {
        perror("Erreur d'ouverture du fichier");
        exit(1);
    }

    printf("Tout est OK...\nOn peut y aller.\n");
    fclose(fb);
    exit(0);
}

FILE *OuvertureFichier(const char *nomFichier,int taille)
{
    FILE *f;

    if ((f = fopen(nomFichier,"r+")) != NULL)
        return f;

    if ((f = fopen(nomFichier,"w+")) == NULL)
        return NULL;

    fseek(f,taille,SEEK_SET);
    fwrite("*",1,1,f);

    return f;
}
```

dont l'exécution fournit :

```
# ls Fich*
ls: impossible d'accéder à Fich*: Aucun fichier ou dossier de ce type
# ./ExempleFonction
Tout est OK...
On peut y aller.
# ls Fich*
Fich.dat
# ./ExempleFonction
Tout est OK...
On peut y aller.
#
```

La fonction OuvertureFichier ouvre un fichier dont le nom est passé en paramètre :

- si ce fichier n'existe pas, il est créé et “bidonné” à une taille précisée par le second paramètre reçu et le caractère * est écrit en fin de fichier. Ce fichier est bidonné “vide” (rien n'a été écrit, donc aucune valeur significative) mais l'espace disque a été réservé pour lui.
- si ce fichier existe, il est ouvert en lecture/écriture.

Dans les deux cas, la fonction retourne un pointeur FILE* non nul vers le buffer de fichier ouvert (voir chapitre sur les fichiers). En cas de problème (plus d'espace disque par exemple), la fonction retourne NULL et **errno** est positionné.

On remarque que les **conventions** d'écriture sont respectées :

- la fonction n'est pas de type int mais de type **FILE***. Lorsque la fonction se déroule correctement, elle retourne un pointeur non nul. En cas d'erreur, le pointeur NULL est retourné.
- la fonction possède **2** paramètres.
- la variable **f** utilisée pour créer/ouvrir le fichier est locale à la fonction. Elle est retournée en cas de succès.
- la fonction ne réalise **aucun affichage** de la cause de l'erreur. En cas d'erreur, fopen positionne la variable **errno** mais celle-ci n'est pas affichée et l'erreur n'est pas traitée dans la fonction. Le traitement de l'erreur est réalisé par la fonction appelante (c'est-à-dire le main ici) en affichant la cause de l'erreur grâce à l'appel de **perror** et en terminant le programme. En effet, le fichier n'ayant pu être créé ou ouvert, le programme n'a pas de raison de continuer ici.

Important ! Une dernière remarque est à faire sur le **stockage** et le **test** de la **valeur de retour** de fopen. Ces deux opérations se font sur la même ligne de code (dans le if) et il est nécessaire de positionner des **parenthèses supplémentaires** à cause de l'**ordre des opérations = et !=** (ou ==). Les opérateurs de comparaison sont en effet prioritaires sur l'opérateur d'affectation. ■

2.2.3 Redirection des entrée/sorties standards

Afin de communiquer avec l'extérieur (et en particulier avec le terminal dans lequel il est lancé), un programme dispose des 3 entrée/sorties standards :

1. **stdin** : pour l'entrée,
2. **stdout** : pour la sortie,
3. **stderr** : pour la sortie des erreurs.

En fait, **stdin**, **stdout** et **stderr** sont des variables (implicites et définies dès l'inclusion du fichier **stdio.h**) de type **FILE*** et peuvent être vues comme des **fichiers bufferisés**. Par défaut,

ces trois fichiers existent et sont **associés au terminal** dans lequel le programme est lancé. Il est possible de manipuler implicitement ou explicitement ces fichiers. Par exemple,

- **printf** écrit implicitement sur le standard de sortie **stdout** (il est en effet inutile de préciser stdout dans les paramètres de printf).
- **perror** écrit implicitement sur le standard d'erreur **stderr** (il est en effet inutile de préciser stderr dans les paramètres de perror).

Pour pouvoir manipuler explicitement ces fichiers, on peut utiliser la fonction **fprintf** qui fait partie de la même famille que printf :

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Quelques explications sur ces fonctions :

- **fprintf** réalise exactement le même travail que printf sauf que l'on peut choisir le "fichier" stream sur lequel on veut écrire.
- **printf** est en réalité un appel de fprintf dont le premier paramètre est stdout.
- **sprintf** réalise à nouveau exactement le même travail que printf sauf que l'écriture se fait dans un **buffer mémoire** dont str est l'adresse.
- **snprintf** réalise le même travail que sprintf mais le nombre de caractères écrits dans le buffer pointé par str est **limité** à size.

Exemple 2.17 (sprintf et snprintf). Soit le programme suivant :

```
/* Test sprintf.cpp */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int aX = 3;
    char nomVar[8];

    strcpy(nomVar, "aX");

    char buffer[80];

    sprintf(buffer, "La valeur de la variable %s est %d", nomVar, aX);
    printf("buffer (%d) = ---%s---\n", strlen(buffer), buffer);

    snprintf(buffer, 20, "La valeur de la variable %s est %d", nomVar, aX);
    printf("buffer (%d) = ---%s---\n", strlen(buffer), buffer);
```

```
    exit(1);
}
```

dont l'exécution fournit :

```
# ./Test_snprintf
buffer (33) = ---La valeur de la variable aX est 3---
buffer (19) = ---La valeur de la var---
#
```

Le second paramètre de **snprintf** limite le remplissage du buffer à 20 caractères. La taille de la chaîne de caractères est donc de 19 étant donné que le dernier est occupé par le caractère '\0'.



Il est possible de rediriger les sorties standards d'un programme vers un **fichier texte**, et cela en ligne de commande. Pour cela, il faut utiliser

- le symbole **>** ou **1>** pour rediriger le **stdout** vers un fichier,
- le symbole **2>** pour rediriger le **stderr** vers un fichier.

Exemple 2.18 (fprintf et redirection des sorties standards). Soit le programme suivant :

```
***** Test_fprintf.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    printf("Premiere écriture sur le stdout\n");
    fprintf(stdout,"Seconde écriture sur le stdout\n");

    errno = 2;
    perror("Premiere écriture sur le stderr");
    fprintf(stderr,"Seconde écriture sur le stderr\n");

    exit(1);
}
```

dont quelques exemples d'exécution fournissent :

```
# ./Test_fprintf
Premiere ecriture sur le stdout
Seconde ecriture sur le stdout
Premiere ecriture sur le stderr: No such file or directory
Seconde ecriture sur le stderr
# ./Test_fprintf > fichier.txt
Premiere ecriture sur le stderr: No such file or directory
Seconde ecriture sur le stderr
# cat fichier.txt
Premiere ecriture sur le stdout
Seconde ecriture sur le stdout
# ./Test_fprintf 1> fichier.txt
Premiere ecriture sur le stderr: No such file or directory
Seconde ecriture sur le stderr
# cat fichier.txt
Premiere ecriture sur le stdout
Seconde ecriture sur le stdout
# ./Test_fprintf 2> fichier.txt
Premiere ecriture sur le stdout
Seconde ecriture sur le stdout
# cat fichier.txt
Premiere ecriture sur le stderr: No such file or directory
Seconde ecriture sur le stderr
# ./Test_fprintf 1> fichier1.txt 2> fichier2.txt
# cat fichier1.txt
Premiere ecriture sur le stdout
Seconde ecriture sur le stdout
# cat fichier2.txt
Premiere ecriture sur le stderr: No such file or directory
Seconde ecriture sur le stderr
#
#
```

Lors de la première exécution du programme, les sorties standards ne sont pas redirigées et tous les affichages se font dans le terminal associé. On remarque également l'équivalence entre **>** et **1>**. Lors de la dernière exécution, on voit que l'on peut **rediriger simultanément** les sorties standard et standard d'erreur vers deux fichiers distincts.



Il est également possible de rediriger l'entrée standard **stdin** d'un programme vers un **fichier texte**, et cela en ligne de commande. Pour cela, il faut utiliser le symbole **<**.

Exemple 2.19 (redirection de l'entrée standard). Soit le programme suivant :

```
***** Test_scanf.cpp *****
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;
    char buffer[40];

    scanf("%d",&a);
    scanf("%s",buffer);
```

```

printf("Lu : --%s-- (%d)\n",buffer,a);

exit(1);
}

```

dont quelques exemples d'exécution fournissent :

```

# ./Test_scanf
50
abcd
Lu : --abcd-- (50)
# cat fichierStdin.txt
12
wagner
# ./Test_scanf < fichierStdin.txt
Lu : --wagner-- (12)
#

```

Lors de la première exécution du programme, l'entrée standard **stdin** n'a pas été redirigée et les valeurs 50 et "abcd" ont été saisies au clavier par l'utilisateur. Lors de la seconde exécution, on voit que **stdin** a été redirigée vers le fichier texte **fichierStdin.txt** et les valeurs 12 et "wagner" ont été lues dans le fichier, sans intervention de l'utilisateur.



2.2.4 Ecriture d'un fichier de traces

Lors de l'exécution d'un programme, il faut toujours pouvoir suivre l'évolution de son exécution (ce que l'on appelle les "**traces**"), et aussi connaître les causes d'erreurs éventuelles ("**traçage des erreurs**"). A la place d'afficher dans le terminal toutes les traces ainsi que l'exécution normale du programme, il serait intéressant que l'ensemble des traces se retrouvent dans un fichier texte qui pourra être analysé ultérieurement. Dès lors, l'idée générale est de

1. réaliser les **sorties "normales "** du programme sur la sortie standard **stdout**, celle-ci n'étant pas redirigée vers un fichier.
2. réaliser les **traces** et les **traces d'erreurs** sur la sortie standard d'erreur **stderr** redirigée vers un fichier texte ("traces.log", "erreurs.log", ...).

Le traçage des erreurs peut alors se réaliser en utilisant la fonction **perror** tandis que le traçage peut se faire à l'aide d'une fonction de "traçage" telle que celle de l'exemple suivant.

Exemple 2.20 (Fichier de traces). Soit le programme suivant :

```

***** Test_Trace.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdarg.h>

void Trace(const char *, ...);

int main()

```

```

{
    printf("Deroulement normal du programme...\n");

    errno = 6;
    perror("Une erreur s'est produite");
    Trace("La valeur de errno est %d",errno);

    exit(0);
}

void Trace(const char *message, ... )
{
    char buffer[80];
    va_list arg;

    va_start(arg,message);
    vsprintf(buffer,message,arg);
    fprintf(stderr,"%s\n",buffer);
    va_end(arg);
}

```

dont l'exécution fournit :

```

# ./Test_Trace 2> traces.log
Deroulement normal du programme...
# cat traces.log
Une erreur s'est produite: No such device or address
La valeur de errno est 6
#

```

La redirection de **stderr** se réalise ici en ligne de commande (avec **2>**) mais on verra plus tard que l'on peut le faire en programmation. On remarque que la fonction **Trace** s'utilise exactement comme **printf** sauf qu'elle écrit sur le **stderr** tandis que **printf** écrit sur le **stdout**. Maintenant, on peut également directement utiliser la fonction **fprintf(stderr,...)**, le résultat serait le même.



2.2.5 Passage d'arguments à un programme : argc et argv

Il est possible de passer des arguments à un programme lancé en **ligne de commande**. Ces arguments seront récupérés sous la forme d'un **tableau de chaînes de caractères**. Le nombre d'éléments de ce tableau est passé en premier argument (**argc**) de la fonction **main()**, et le tableau (**argv[]**) est transmis en second argument.

La première position (indice 0) du tableau contient toujours le **nom de l'exécutable**. Dès lors, **argc** est égal au **nombre de "mots"** (ou "item" ou encore "token") sur la ligne de commande, y compris le nom de l'exécutable. De plus, le tableau **argv[]** contient (**argc+1**) éléments, le dernier étant un pointeur **NULL**.

Exemple 2.21 (Passage d'arguments à un programme). Soit le programme suivant :

```
***** Test_argv.cpp *****
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
    int i=0;

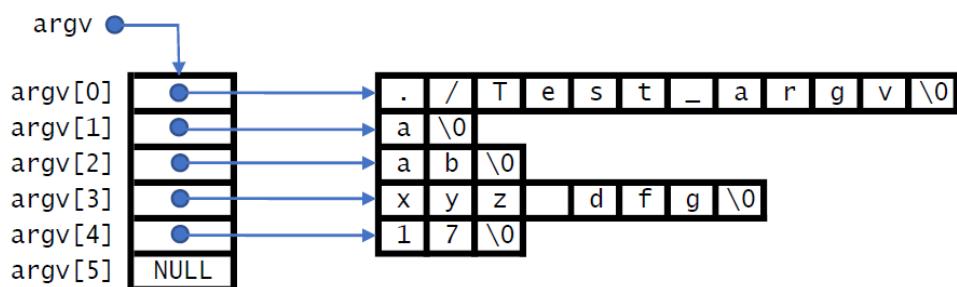
    printf("argc = %d\n",argc);
    printf("%s a recu en parametre :\n",argv[0]);
    for (i=1 ; i<argc ; i++)
        printf("--%s--\n",argv[i]);
    printf("Dernier argument sous forme de int : %d\n",atoi(argv[i-1]));

    exit(0);
}
```

dont un exemple d'exécution fournit :

```
# ./Test_argv a ab "xyz dfg" 17
argc = 5
./Test_argv a recu en parametre :
--a--
--ab--
--xyz dfg--
--17--
Dernier argument sous forme de int : 17
#
```

La situation est représentée à la figure suivante :



On remarque que

1. si on veut passer en argument une chaîne de caractères contenant des **espaces vides**, cet argument doit être passé entre double quotes ("").
2. il est **impossible** de recevoir directement un **int** (ou float, etc...) en ligne de commande. On ne peut récupérer que des chaînes de caractères. De là, il est nécessaire d'utiliser une fonction du type **atoi** (ou atof, etc...) pour réaliser la conversion.



2.3 Ecriture d'une application en plusieurs fichiers

Une application complexe peut contenir plusieurs programmes donnant naissance à plusieurs exécutables. De plus, ces différents programmes peuvent comporter et utiliser les mêmes fonctions, structures, etc. Jusque maintenant, les fonctions utilisées dans un programme étaient déclarées et définies dans le même fichier. Dès lors, il se peut que certaines fonctions soient recopiées intégralement dans plusieurs fichiers, une modification d'un fichier n'entrant pas automatiquement la modification des autres fichiers.

Nous allons voir comment structurer une application en **plusieurs fichiers** afin d'éviter cette redondance et permettre une meilleure maintenance. Une fonction modifiée dans un fichier sera alors modifiée pour tous les programmes qui l'utilisent.

Considérons le programme

```
***** Applic1.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int minimum(int x,int y);
void affiche(const char *m);

int main()
{
    char buffer[80];

    sprintf(buffer,"Le minimum de 7 et 3 est %d",minimum(7,3));
    affiche(buffer);

    exit(1);
}

int minimum(int x,int y)
{
    if (x < y) return x;
    return y;
}

void affiche(const char *m)
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}
```

qui comporte 2 fonctions, l'une (minimum) permettant de réaliser une opération mathématique et l'autre (affiche) permettant d'afficher un message précédé de la date actuelle, et le programme

```
***** Applic2.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void affiche(const char *m);
```

```

int main()
{
    affiche("Bonjour !");
    exit(1);
}

void affiche(const char *m)
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}

```

qui comporte la même fonction (affiche) que le programme précédent. Il y a donc une partie commune aux 2 programmes. Si on souhaite modifier la fonction affiche, il faut le faire dans les 2 fichiers, ce qui devient vite très lourd dès que la fonction est répétée à plusieurs endroits et que la modification doit se faire partout.

La compilation et l'exécution de ces deux programmes fournit

```

# g++ Applic1.cpp -o Applic1
# g++ Applic2.cpp -o Applic2
# ./Applic1
[29/4/2021] Le minimum de 7 et 3 est 3
# ./Applic2
[29/4/2021] Bonjour !
#

```

2.3.1 Crédit des fichiers .cpp

L'idée est alors de découper l'application en "blocs" et idéalement de rassembler les fonctions par thèmes. Dans notre cas, il y a deux thèmes

1. **Mathématique** : on va créer un fichier **LibMath.cpp** qui va contenir la définition de la fonction minimum,
2. **Entrées/sorties** : on va créer un fichier **LibIO.cpp** qui va contenir la définition de la fonction affiche.

Les fichiers créés sont donc

```

/***** LibMath.cpp *****/
int minimum(int x,int y)
{
    if (x < y) return x;
    return y;
}

```

et

```

/***** LibIO.cpp *****/
#include <stdio.h>
#include <time.h>

void affiche(const char *m)

```

```
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}
```

On peut donc imaginer de modifier les fichiers Applic1.cpp et Applic2.cpp de la manière suivante :

```
***** Applic1.cpp *****/
#include <stdio.h>
#include <stdlib.h>

#include "LibIO.cpp"
#include "LibMath.cpp"

int main()
{
    char buffer[80];
    sprintf(buffer,"Le minimum de 7 et 3 est %d",minimum(7,3));
    affiche(buffer);

    exit(1);
}
```

et

```
***** Applic2.cpp *****/
#include <stdlib.h>

#include "LibIO.cpp"

int main()
{
    affiche("Bonjour !");

    exit(1);
}
```

La compilation et l'exécution des deux programmes restent identiques.

Etant donné la directive **#include** utilisée pour inclure les fichiers LibIO.cpp et LibMath.cpp, on remarque que si une modification de la fonction affiche est réalisée dans le fichier LibIO.cpp, elle est répercutée automatiquement sur les deux programmes. Cependant, cette manière de faire pose plusieurs problèmes :

1. La compilation de Applic1.cpp et de Applic2.cpp impliquent que le fichier LibIO.cpp est **compilé deux fois**.
2. Une modification du main et dès lors une recompilation d'un ou des deux programme(s) impliquent forcément la **recompilation** des fonctions utilisées, **même** si celles-ci n'ont **pas été modifiées**.

La bonne méthode est alors de compiler séparément chaque fichier .cpp et de les assembler lors de l'édition de liens.

Tentons de compiler le fichier LibIO.cpp :

```
# g++ LibIO.cpp
/lib/.../lib64/crt1.o : Dans la fonction « _start » :
(.text+0x20) : référence indéfinie vers « main »
collect2: error: ld returned 1 exit status
# g++ LibIO.cpp -c
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.o LibMath.cpp
#
```

On remarque que

1. la première compilation n'a pas abouti. Il y a une erreur d'édition de liens (**ld**). En effet, le compilateur cherche à créer un fichier exécutable mais le fichier LibIO.cpp ne comporte **pas de fonction main**.
2. lors de la seconde compilation, l'option **-c** a été utilisée afin de stopper la compilation juste avant l'édition de liens, ce qui produit le fichier **LibIO.o** qui est un fichier binaire, résultat de la compilation de la fonction affiche.

Nous pouvons à présent faire de même pour le fichier LibMath.cpp et tenter de réaliser l'édition de liens :

```
# g++ LibMath.cpp -c
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.o LibMath.cpp LibMath.o
# g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
LibIO.o : Dans la fonction « affiche(char const*) » :
LibIO.cpp:(.text+0x0) : définitions multiples de « affiche(char const*) »
/tmp/ccxEiINH.o:Applic1.cpp:(.text+0x0) : défini pour la première fois ici
LibMath.o : Dans la fonction « minimum(int, int) » :
LibMath.cpp:(.text+0x0) : définitions multiples de « minimum(int, int) »
/tmp/ccxEiINH.o:Applic1.cpp:(.text+0x63) : défini pour la première fois ici
collect2: error: ld returned 1 exit status
#
```

On remarque qu'il y a à nouveau un problème d'édition de liens (**ld**). Les fonctions affiche et minimum sont **définies** dans les fichiers LibIO.o et LibMath.o et **redéfinies** dans Applic1.cpp.

Supprimons alors les deux **#include** dans le fichier Applic1.cpp :

```
***** Applic1.cpp ****/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char buffer[80];
    sprintf(buffer, "Le minimum de 7 et 3 est %d", minimum(7,3));
    affiche(buffer);

    exit(1);
}
```

Si on tente à nouveau la compilation, cela donne :

```
# g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
Applic1.cpp: In function ‘int main()’:
Applic1.cpp:9:59: error: ‘minimum’ was not declared in this scope
    sprintf(buffer,"Le minimum de 7 et 3 est %d",minimum(7,3));
                                         ^
Applic1.cpp:10:17: error: ‘affiche’ was not declared in this scope
    affiche(buffer);
               ^
#
#
```

On remarque qu'il y a à présent une **erreur de compilation**. Bien que les fonctions affiche et minimum soient définies dans les fichiers LibIO.o et LibMath.o, elles ne sont **pas déclarées** dans Applic1.cpp. En effet, les **prototypes** de ces fonctions ne sont **pas connus** dans Applic1.cpp.

2.3.2 Crédit des fichiers .h

Les prototypes manquants des fonctions affiche et minimum vont être placés dans des **fichiers "headers"**, d'extension **.h**. Nous allons donc créer les fichiers

1. **LibMath.h** qui va contenir le **prototype** de la fonction minimum,
2. **LibIO.h** qui va contenir le **prototype** de la fonction affiche.

Chaque module sera alors composé d'un fichier .cpp et d'un fichier .h. Le module "mathématiques" va être composé du fichier

```
***** LibMath.h ****/
#ifndef LIB_MATH
#define LIB_MATH

int minimum(int x,int y);

#endif
```

et du fichier

```
***** LibMath.cpp ****/
#include "LibMath.h"

int minimum(int x,int y)
{
    if (x < y) return x;
    return y;
}
```

tandis que le module "entrées/sorties" va être composé du fichier

```
***** LibIO.h ****/
#ifndef LIB_IO
#define LIB_IO

void affiche(const char *m);

#endif
```

et du fichier

```
***** LibIO.cpp *****
#include "LibIO.h"

#include <stdio.h>
#include <time.h>

void affiche(const char *m)
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}
```

On remarque que

1. le fichier LibIO.cpp (LibMath.cpp) inclut le fichier LibIO.h (LibMath.h). Dans notre exemple, ce n'est pas indispensable mais cela devient nécessaire quand le fichier .h contient la déclaration de structures utilisées par les fonctions du module ou lorsqu'une fonction du module appelle un autre fonction du même module.
2. Les directives **#ifndef... #define... #endif** sont utilisées dans les fichiers .h afin d'**éviter d'inclure plusieurs fois les prototypes des fonctions** (ou les structures déclarées) si jamais le même fichier header était inclus plusieurs fois par l'intermédiaire de plusieurs modules. Le nom de la **macro** utilisée doit être unique et propre au module et est de préférence constitué de majuscules (afin d'éviter des confusions avec des variables).

Gra e aux fichiers headers ainsi cr  s, on peut modifier Applic1.cpp et Applic2.cpp selon

```
***** Applic1.cpp *****
#include <stdio.h>
#include <stdlib.h>

#include "LibMath.h"
#include "LibIO.h"

int main()
{
    char buffer[80];

    sprintf(buffer,"Le minimum de 7 et 3 est %d",minimum(7,3));
    affiche(buffer);

    exit(1);
}
```

et

```
***** Applic2.cpp *****
#include <stdlib.h>

#include "LibIO.h"

int main()
{
    affiche("Bonjour !");

    exit(1);
}
```

La compilation et l'édition de liens peuvent à présent être réalisées sans problème :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h
# g++ LibIO.cpp -c
# g++ LibMath.cpp -c
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp LibMath.h LibMath.o
# g++ Applic1.cpp LibMath.o LibIO.o -o Applic1
# g++ Applic2.cpp LibIO.o -o Applic2
# ./Applic1
[29/4/2021] Le minimum de 7 et 3 est 3
# ./Applic2
[29/4/2021] Bonjour !
#
```

On remarque que

1. lors de la création de l'exécutable Applic2, l'édition de liens n'a pas besoin du fichier LibMath.o. En effet, Applic2.cpp n'utilise pas la fonction minimum.
2. la recompilation de Applic1.cpp ou de Applic2.cpp ne nécessite pas de modification des fichiers LibMath.o et LibIO.o.

2.3.3 Le mot clé “extern”

Si une **variable globale** doit être **utilisée dans plusieurs fichiers .cpp** et qu'elle est déclarée globale dans chacun des fichiers, il n'y aura aucun problème lors de la création des fichiers .o mais bien lors de l'**édition de liens**. En effet, plusieurs espaces mémoires seraient réservés pour la même variable, ce qui n'est pas possible.

Remarquons qu'il ne faut pas réserver d'espaces mémoires (en déclarant une variable globale par exemple) dans un fichier .h, sinon cet espace mémoire pourrait être réservé plusieurs fois en cas de multiples inclusions de ce fichier.

Reprendons les fichiers Applic2.cpp et LibIO.cpp et déclarons-y une variable globale a de type int :

```
***** Applic2.cpp *****/
#include <stdlib.h>

#include "LibIO.h"

int a;

int main()
{
    affiche("Bonjour !");
    exit(1);
}
```

et

```
***** LibIO.cpp *****/
```

```
#include "LibIO.h"

#include <stdio.h>
#include <time.h>

int a;

void affiche(const char *m)
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}
```

dont la compilation et l'édition de liens fournissent :

```
# g++ LibIO.cpp -c
# g++ Applic2.cpp LibIO.o -o Applic2
LibIO.o:(.bss+0x0) : définition multiples de « a »
/tmp/ccak2HH1.o:(.bss+0x0) : défini pour la première fois ici
collect2: error: ld returned 1 exit status
#
```

Pour remédier à ce problème, il faut utiliser le mot clé **extern** et déclarer la variable globale **normalement dans un fichier et en extern dans tous les autres**. Ainsi, lors de l'édition de liens, la variable ne sera **réservée qu'une seule fois en mémoire**.

Modifions à nouveau le fichier LibIO.cpp :

```
***** LibIO.cpp *****/
#include "LibIO.h"

#include <stdio.h>
#include <time.h>

extern int a;

void affiche(const char *m)
{
    time_t temps = time(NULL);
    struct tm * ptr = localtime(&temps);
    printf("[%d/%d/%d] %s\n",ptr->tm_mday,ptr->tm_mon+1,ptr->tm_year+1900,m);
}
```

La compilation et l'édition de liens fournissent à présent :

```
# g++ LibIO.cpp -c
# g++ Applic2.cpp LibIO.o -o Applic2
#
```

Remarquons cependant que cette variable globale ne pourra être **initialisée que dans un seul des fichiers**, ce qui est logique. Par exemple, si `a` est initialisée à 3 dans `LibIO.cpp` et à 5 dans `Applic2.cpp`, la compilation et l'édition de liens fournissent :

```
# g++ LibIO.cpp -c
LibIO.cpp:7:12: warning: 'a' initialized and declared 'extern'
extern int a = 3;
^

# g++ Applic2.cpp LibIO.o -o Applic2
LibIO.o:(.data+0x0) : définition multiples de « a »
/tmp/ccUHzLDn.o:(.data+0x0) : défini pour la première fois ici
collect2: error: ld returned 1 exit status
#
```

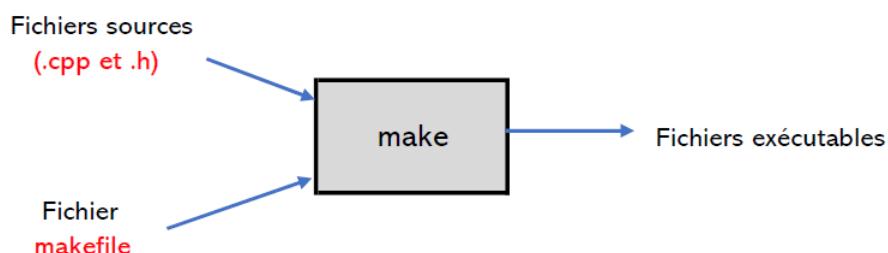
2.4 Automatisation de la compilation : la commande make

Dès qu'une application s'appuie sur plusieurs modules indépendants (plusieurs fichiers sources `.cpp` et `.h`), il devient vite fastidieux de compiler manuellement chaque module et de réaliser l'édition de liens pour chaque exécutable. De plus, lors du développement, il n'est pas toujours nécessaire de recompiler l'entièreté des fichiers. Il est donc indispensable de disposer d'un mécanisme **automatique** de compilation permettant

1. une génération automatique des fichiers objets `.o` et des exécutables de l'application,
2. de recompiler uniquement les fichiers récemment modifiés et donc de se limiter à l'édition de liens pour les exécutables dépendant de ces fichiers.

La commande **make** permet de générer une application d'après une description faite dans un **fichier texte** appelé **makefile** (ou **Makefile**). Cette génération est réalisée de façon optimale, c'est-à-dire que seules les opérations concernant les fichiers ayant subi une modification sont effectuées. Ce travail se fait évidemment en comparant les dates de modification des différents fichiers intervenants.

Schématiquement :



Le fichier `makefile` est un **fichier de configuration** généralement situé dans le même répertoire que les fichiers sources. Il est structuré selon une série de règles de la forme

```
cible : dépendance1 dépendance2 ... dépendanceN
      commandes
```

où

1. la **cible** est le fichier que l'on désire créer,

2. les **dépendances** sont les fichiers nécessaires à la création du fichier cible,
3. **commandes** sont les commandes (compilation, copie, ...) nécessaires pour créer le fichier cible à partir des dépendances.

La commande **make** reçoit en argument le fichier makefile et compare les dates de modification des fichiers sources (les dépendances) et du fichier cible (s'il existe déjà) pour savoir si les commandes correspondantes doivent être exécutées.

2.4.1 Un premier makefile

Reprendons l'exemple d'application utilisé précédemment pour décrire la compilation en fichiers séparés et dont le résultat final est constitué des deux exécutables **Applic1** et **Applic2**. Nous disposons donc de l'ensemble des fichiers sources et la compilation fournit :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h
# g++ LibIO.cpp -c
# g++ LibMath.cpp -c
# g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
# g++ Applic2.cpp LibIO.o -o Applic2
# ./Applic1
[4/5/2021] Le minimum de 7 et 3 est 3
# ./Applic2
[4/5/2021] Bonjour !
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp
LibMath.h LibMath.o
#
```

Notons dès à présent que si les différents fichiers ne compilent pas correctement, l'utilisation d'un makefile et de la commande **make** ne résoudra pas les problèmes de compilation.

L'observation des 4 lignes de compilation nous montre que

1. Le produit final du processus de compilation est constitué des fichiers exécutables **Applic1** et **Applic2**. Ils vont constituer les **cibles principales** du makefile.
2. Les fichiers **LibIO.o** et **LibMath.o** ne sont pas les produits finaux du processus de compilation mais sont nécessaires à la création des exécutables Applic1 et Applic2. Ils sont donc des **cibles secondaires** du makefile et des **dépendances** des cibles principales Applic1 et Applic2.
3. Le fichier LibIO.o (LibMath.o) dépend des fichiers LibIO.cpp (LibMath.cpp) et LibIO.h (LibMath.h). En effet, la modification d'un des deux nécessite la recréation du fichier objet.

Ces constatations mènent à la création du fichier **makefile** suivant :

```
# Fichier makefile (ligne de commentaires)
Applic1:    Applic1.cpp LibIO.o LibMath.o
                g++ Applic1.cpp LibIO.o LibMath.o -o Applic1

Applic2:    Applic2.cpp LibIO.o
                g++ Applic2.cpp LibIO.o -o Applic2

LibIO.o:    LibIO.cpp LibIO.h
                g++ LibIO.cpp -c

LibMath.o:  LibMath.cpp LibMath.h
                g++ LibMath.cpp -c
```

La syntaxe d'un fichier makefile est assez pénible. En effet, pour chaque cible,

1. l'espace entre : et la première dépendance doit être une **tabulation**,
2. sur la ligne de commande, l'espace avant g++ (ou une autre commande selon le cas), est constitué d'une ou plusieurs **tabulations**, mais **jamais d'espace**.

L'utilisation du fichier makefile peut alors se faire en ligne de commande :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile
# make
g++ LibIO.cpp -c
g++ LibMath.cpp -c
g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
# make
make: « Applic1 » est à jour.
# ls
Applic1 Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp LibMath.h
LibMath.o makefile
# ./Applic1
[5/5/2021] Le minimum de 7 et 3 est 3
#
```

On peut remarquer que

1. Lors de la **première exécution** de make, les fichiers LibIO.o et LibMath.o ont été créés car ce sont les dépendances de la cible principale Applic1. Le fichier Applic1.cpp est également une dépendance mais c'est un fichier source trouvé par la commande make dans le répertoire courant.
2. Lors de la **seconde exécution** de make, rien n'a été recompilé. En effet, entre les deux exécutions de make, aucun fichier n'a été modifié et la date de dernière modification du fichier Applic1 est postérieure à celles de toutes ses dépendances.
3. Plus important : seul l'exécutable Applic1 a été créé. En effet,

La vie d'un fichier makefile est de créer **uniquement sa première cible**, appelée **cible principale** du makefile. Dès lors, seules les cibles dépendant de la cible principale sont créées.

Dans notre cas, vu que Applic2 n'est pas une dépendance de Applic1 (cible principale du makefile), il n'est pas créé ! En effet, la commande make est incapable de savoir que Applic2 est un des résultats attendus pour nous.

Remarque : Choix d'une cible particulière

La commande make permet de créer une cible que l'on désire créer en particulier. Pour cela, il suffit de lui passer cette cible en argument.

On pourrait par exemple imaginer ceci :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile
# make Applic2
g++ LibIO.cpp -c
g++ Applic2.cpp LibIO.o -o Applic2
# ls
Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp LibMath.h
makefile
# make LibMath.o
g++ LibMath.cpp -c
# ls
Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp LibMath.h
LibMath.o makefile
#
```

Remarque : Un nom de fichier différent pour le fichier makefile

Lorsque la commande make est utilisée sans paramètre, elle cherche dans le répertoire courant un fichier qui s'appelle “**makefile**” ou “**Makefile**”. Il est cependant possible de travailler avec un fichier dont le nom est différent. Il faut alors utiliser l'option **-f** de la commande make.

Considérons le fichier makefile dont le nom est “**MyMake**” :

```
# MyMake
Applic2: Applic2.cpp LibIO.o
          g++ Applic2.cpp LibIO.o -o Applic2

LibIO.o: LibIO.cpp LibIO.h
          g++ LibIO.cpp -c
```

Celui-ci peut être utilisé en ligne de commande :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile MyMake
# make -f MyMake
g++ LibIO.cpp -c
g++ Applic2.cpp LibIO.o -o Applic2
# ls
Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp LibMath.h
makefile MyMake
#
```

2.4.2 Crédation de plusieurs exécutables avec le même fichier makefile

Pour créer plusieurs fichiers exécutables, il n'est pas nécessaire de créer autant de fichiers makefile qu'il y a d'exécutables. Il suffit d'ajouter au début du fichier makefile une nouvelle cible, qui va donc devenir la **cible principale** du makefile, dont les dépendances sont les fichiers exécutables que l'on désire créer.

Dans notre cas, le makefile devient :

```
# Fichier makefile  (ligne de commentaires)
all:      Applic1 Applic2

Applic1:  Applic1.cpp LibIO.o LibMath.o
          g++ Applic1.cpp LibIO.o LibMath.o -o Applic1

Applic2:  Applic2.cpp LibIO.o
          g++ Applic2.cpp LibIO.o -o Applic2

LibIO.o:  LibIO.cpp LibIO.h
          g++ LibIO.cpp -c

LibMath.o: LibMath.cpp LibMath.h
           g++ LibMath.cpp -c
```

La **cible principale** du makefile est à présent “**all**” (le nom de la cible n'a aucune importance). Pour être créée, elle a besoin des fichiers Applic1 et Applic2 qui vont être créés en tant que cibles secondaires du makefile. Une fois ces deux fichiers créés, le travail du makefile est terminé. D'ailleurs, il n'y a aucune commande associée à la cible **all**.

L'utilisation de la commande make fournit alors :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile
# make
g++ LibIO.cpp -c
g++ LibMath.cpp -c
g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
g++ Applic2.cpp LibIO.o -o Applic2
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp
LibMath.h LibMath.o makefile
# rm LibMath.o
# make
g++ LibMath.cpp -c
g++ Applic1.cpp LibIO.o LibMath.o -o Applic1
# touch Applic2.cpp
# make
g++ Applic2.cpp LibIO.o -o Applic2
#
```

Nous remarquons à présent que

1. Lors de la première exécution de make, tous les fichiers ont été créés, y compris l'exécutable **Applic2**.
2. Lors de la seconde exécution de make, le fichier LibMath.o ayant été supprimé, il est recréé et seul l'exécutable Applic1, qui seul en dépend, est recréé.
3. La commande **touch** modifie la date de dernière modification du fichier passé en argument. Dès lors, lors de la troisième exécution de make, la date de Applic2 est antérieure à celle de Applic2.cpp et l'exécutable Applic2 est recréé.

2.4.3 Paramétrisation d'un fichier makefile

Il est possible d'intégrer à un fichier makefile plusieurs éléments qui permettent de le rendre plus ergonomique et paramétrable, notamment

- les **commentaires** : toute ligne commençant par le caractère **#** est considérée comme une ligne de commentaires.
- les **variables** : un makefile peut comporter des variables dont le contenu ne peut être qu'une **chaîne de caractères**. Pour affecter le contenu d'une variable, il faut utiliser l'opérateur **=** tandis que pour manipuler le contenu de cette variable, il faut la faire précédé par le caractère **\$**.
- la **suppression de l'écho** des commandes de compilation soit par l'option **--silent** de la commande make, soit par la cible particulière **.SILENT** : ajoutée au début du fichier makefile.

En guise d'exemple, modifions notre makefile afin de paramétrier le compilateur et ses options :

```
# Fichier makefile (avec variables de makefile)
COMP=g++ -Wall -D DEBUG

all:      Applic1 Applic2

Applic1:  Applic1.cpp LibIO.o LibMath.o
          $(COMP) Applic1.cpp LibIO.o LibMath.o -o Applic1

Applic2:  Applic2.cpp LibIO.o
          $(COMP) Applic2.cpp LibIO.o -o Applic2

LibIO.o:  LibIO.cpp LibIO.h
          $(COMP) LibIO.cpp -c

LibMath.o: LibMath.cpp LibMath.h
          $(COMP) LibMath.cpp -c
```

La variable de makefile **COMP** (par convention, il est préférable de mettre toutes les lettres d'une variable de makefile en majuscule) contient la commande de compilation avec ses options. Lors de l'exécution de la commande make, **\$(COMP)** est remplacé par sa valeur "g++ -Wall -D DEBUG". Pour ajouter ou supprimer une option de compilation (comme -Wall ou -D DEBUG par exemple) qui doit être prise en compte pour toutes les cibles du makefile, il suffit de modifier la variable **COMP**.

L'utilisation de la commande make fournit alors :

```
# ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile
# make
g++ -Wall -D DEBUG LibIO.cpp -c
g++ -Wall -D DEBUG LibMath.cpp -c
g++ -Wall -D DEBUG Applic1.cpp LibIO.o LibMath.o -o Applic1
g++ -Wall -D DEBUG Applic2.cpp LibIO.o -o Applic2
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp
LibMath.h LibMath.o makefile
# rm *.o; rm Applic1; rm Applic2; ls
Applic1.cpp Applic2.cpp LibIO.cpp LibIO.h LibMath.cpp LibMath.h makefile
# make --silent
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO.cpp LibIO.h LibIO.o LibMath.cpp
LibMath.h LibMath.o makefile
#
```

Nous remarquons que

1. La variable **COMP** a bien été remplacée par sa valeur pour la création de chacune des cibles.
2. L'option **--silent** a bien supprimé l'écho, aucune ligne de compilation n'apparaît sur la console alors que la compilation a bien été réalisée.

Organisation en sous-réertoires

Lorsqu'une application est constituée de nombreux fichiers (.h et .cpp), il est plus approprié de créer des **sous-réertoires** afin d'y ranger les fichiers par thèmes. Plusieurs possibilités existent :

1. Créer un répertoire par module contenant le fichier source .cpp accompagné de son fichier header .h
2. Créer un répertoire **sources** contenant les fichiers .cpp, un répertoire **includes** contenant les fichiers .h, un répertoire **objs** contenant les fichiers objets .o et un fichier **bin** contenant les fichiers exécutables résultant de la compilation.
3. ...

L'architecture choisie ainsi que le nom des répertoires peuvent être quelconques mais dans tous les cas, il est possible d'en tenir compte dans le fichier **makefile**.

Revenons à notre exemple de création des exécutables Applic1 et Applic2, et choisissons de créer un répertoire par module, **LibMath** et **LibIO** qui vont contenir chacun leurs fichiers .cpp et .h. Dans ce cas, la compilation manuelle se réalise selon

```
# ls
Applic1.cpp Applic2.cpp LibIO LibMath
# ls LibIO/
LibIO.cpp LibIO.h
# ls LibMath/
LibMath.cpp LibMath.h
# g++ -c LibIO/LibIO.cpp -I LibIO -o LibIO/LibIO.o
# g++ -c LibMath/LibMath.cpp -I LibMath -o LibMath/LibMath.o
# ls LibIO/
LibIO.cpp LibIO.h LibIO.o
# ls LibMath/
LibMath.cpp LibMath.h LibMath.o
# g++ Applic1.cpp LibIO/LibIO.o LibMath/LibMath.o -I LibIO -I LibMath -o Applic1
# g++ Applic2.cpp LibIO/LibIO.o -I LibIO -o Applic2
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO LibMath
#
```

Nous pouvons remarquer que

1. Pour la création des fichiers objets .o, l'option **-o** est utilisée afin de **positionner directement le fichier** dans son répertoire. Sans ça, les fichiers objets seraient créés dans le répertoire courant.
2. Lors de l'édition de liens, il est nécessaire d'aller rechercher les fichiers objets dans leurs répertoires respectifs.
3. Quelle que soit la ligne de compilation, l'option **-I** a été utilisée afin que les directives **#include** des fichiers .cpp puissent trouver les fichiers headers dans les bons répertoires.

Nous pouvons à présent construire un fichier makefile tenant compte de ces sous-réertoires :

```
# Fichier makefile (avec sous-repertoires)
LIB_IO=./LibIO
LIB_MATH=./LibMath
COMP=g++ -I $(LIB_IO) -I $(LIB_MATH)

all: Applic1 Applic2

Applic1: Applic1.cpp $(LIB_IO)/LibIO.o $(LIB_MATH)/LibMath.o
          $(COMP) Applic1.cpp $(LIB_IO)/LibIO.o $(LIB_MATH)/LibMath.o -o Applic1

Applic2: Applic2.cpp $(LIB_IO)/LibIO.o
          $(COMP) Applic2.cpp $(LIB_IO)/LibIO.o -o Applic2

$(LIB_IO)/LibIO.o: $(LIB_IO)/LibIO.cpp $(LIB_IO)/LibIO.h
                     $(COMP) $(LIB_IO)/LibIO.cpp -c -o $(LIB_IO)/LibIO.o

$(LIB_MATH)/LibMath.o: $(LIB_MATH)/LibMath.cpp $(LIB_MATH)/LibMath.h
                     $(COMP) $(LIB_MATH)/LibMath.cpp -c -o $(LIB_MATH)/LibMath.o
```

Les variables de makefile **LIB_IO** et **LIB_MATH** contiennent le nom des répertoires contenant les fichiers sources de chacun des modules et sont utilisées partout dans le fichier makefile. L'avantage de travailler de la sorte est que si on déplace ou renomme un de ces répertoires, il suffit de changer le contenu de la variable de makefile correspondante.

L'utilisation de ce makefile fournit alors

```
# ls
Applic1.cpp Applic2.cpp LibIO LibMath makefile
# make
g++ -I ./LibIO -I ./LibMath ./LibIO/LibIO.cpp -c -o ./LibIO/LibIO.o
g++ -I ./LibIO -I ./LibMath ./LibMath/LibMath.cpp -c -o ./LibMath/LibMath.o
g++ -I ./LibIO -I ./LibMath Applic1.cpp ./LibIO/LibIO.o ./LibMath/LibMath.o -o
Applic1
g++ -I ./LibIO -I ./LibMath Applic2.cpp ./LibIO/LibIO.o -o Applic2
# ls LibIO/
LibIO.cpp LibIO.h LibIO.o
# ls LibMath/
LibMath.cpp LibMath.h LibMath.o
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO LibMath makefile
#
```

Une fois les variables de makefile remplacées par leur valeurs, on remarque que les lignes de compilation sont générées correctement.

Ajout de cibles utilitaires

Une fois la phase de développement terminée, il pourrait être intéressant de

- Effacer l'**ensemble des fichiers objets .o** qui ne sont en fait que des produits de compilation intermédiaires et non utiles pour la déploiement de l'application. Ceci va donner naissance à une **cible du makefile** que l'on nommera **clean**.

- Effacer l'**ensemble des fichiers compilés**, aussi bien les fichiers objets que les exécutables, pour, par exemple, porter les sources et réaliser la compilation de l'application sur un autre système. Ceci donnera naissance à une **cible du makefile** que l'on nommera **clobber**.

Nous modifions donc notre makefile selon

```
# Fichier makefile  (complet)
.SILENT:
LIB_IO=../LibIO
LIB_MATH=../LibMath
COMP=g++ -I $(LIB_IO) -I $(LIB_MATH)
OBJS=$(LIB_IO)/LibIO.o $(LIB_MATH)/LibMath.o
PROGRAMS=Applic1 Applic2

all: $(PROGRAMS)

Applic1:    Applic1.cpp $(OBJS)
            echo Creation de Applic1
            $(COMP) Applic1.cpp $(OBJS) -o Applic1

Applic2:    Applic2.cpp $(LIB_IO)/LibIO.o
            echo Creation de Applic2
            $(COMP) Applic2.cpp $(LIB_IO)/LibIO.o -o Applic2

$(LIB_IO)/LibIO.o:    $(LIB_IO)/LibIO.cpp $(LIB_IO)/LibIO.h
            echo Creation de LibIO.o
            $(COMP) $(LIB_IO)/LibIO.cpp -c -o $(LIB_IO)/LibIO.o

$(LIB_MATH)/LibMath.o:    $(LIB_MATH)/LibMath.cpp $(LIB_MATH)/LibMath.h
            echo Creation de LibMath.o
            $(COMP) $(LIB_MATH)/LibMath.cpp -c -o $(LIB_MATH)/LibMath.o

clean:
        rm -f $(OBJS) core

clobber:    clean
        rm -f $(PROGRAMS)
```

Nous remarquons que

1. La cible spéciale **.SILENT:** a été ajoutée en début de makefile afin que les lignes de compilation n'apparaissent pas lors de l'exécution de make. Néanmoins, l'utilisation de la commande **echo** permet d'afficher un message pour chaque cible créée.
2. Deux variables de makefile ont été ajoutées : **OBJS** qui contient l'ensemble des fichiers objets de l'application et **PROGRAMS** qui contient l'ensemble des fichiers exécutables de l'application.
3. la cible secondaire **clean** n'a pas de dépendance, elle efface (sans confirmation via l'option **-f** de la commande **rm**) tous les fichiers objets de l'application (grâce à la variable **OBJS**) ainsi que les éventuels fichiers **core** résultant d'une exécution ayant provoqué une erreur de segmentation de mémoire.
4. la cible secondaire **clobber** possède la dépendance **clean**, cela implique que dans un

premier temps, la cible clean est “créée”, ce qui revient à dire que les fichiers objets de l’application et les fichiers core sont supprimés. Dans un second temps, tous les fichiers exécutables sont supprimés (grâce à la variable PROGRAMS).

L’utilisation de ce makefile avec la commande make fournit

```
# ls
Applic1.cpp Applic2.cpp LibIO LibMath makefile
# make
Creation de LibIO.o
Creation de LibMath.o
Creation de Applic1
Creation de Applic2
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO LibMath makefile
# make clean
# ls
Applic1 Applic1.cpp Applic2 Applic2.cpp LibIO LibMath makefile
# ls LibIO/
LibIO.cpp LibIO.h
# ls LibMath/
LibMath.cpp LibMath.h
# make clobber
# ls
Applic1.cpp Applic2.cpp LibIO LibMath makefile
#
```

Il existe bien d’autres possibilités pour optimiser la création de makefile mais cela sort du cadre de ce cours.

Chapitre 3

Les fichiers sous Linux

3.1 Généralités

3.1.1 Utilisateurs et groupes

Sur une machine Linux, chaque utilisateur est identifié par une chaîne de caractères unique. Celle-ci s'appelle le **“User Name”**. A cette chaîne de caractères est associé un entier positif unique, appelé le **“User Id”** (ou **UID**).

Ensuite, les utilisateurs de la machine peuvent être regroupés en fonction de leur rôle. Le système définit donc la notion de groupe d'utilisateurs ou **“Group Id”**. Il s'agit à nouveau d'un entier positif unique (**GID**).

Les utilisateurs et les groupes sont définis dans les fichiers systèmes

1. **/etc/passwd** : pour les utilisateurs
2. **/etc/group** : pour les groupes

Exemple 3.1 (/etc/passwd). Observons le contenu du fichier `/etc/passwd` sur une machine Oracle Linux :

```
# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
...
student:x:1000:1000:Student:/home/student:/bin/bash
...
wagner:x:500:500::/home/wagner:/bin/bash
#
```

Chaque ligne du fichier **/etc/passwd** correspond à un utilisateur et comporte 7 champs :

1. le **“User Name”** (ou le login ou le “nom de connexion”) sous forme d'une chaîne de caractères.
2. Anciennement le **mot de passe** crypté. Dans les systèmes sécurisés récents, ce mot de passe est stocké dans un autre fichier et apparaît ici comme un ‘x’.
3. le **UID** (“User id”) l'entier positif qui identifie l'utilisateur de manière unique.

4. le **GID** ("Group id") l'entier positif qui identifie de manière unique le groupe dans lequel se trouve l'utilisateur.
5. le **commentaire** qui peut être librement utilisé par l'administrateur.
6. le **répertoire de connexion** de l'utilisateur.
7. le **shell** ou l'interpréteur de commandes obtenu lors de la connexion.

Exemple 3.2 (/etc/group). Observons le contenu du fichier /etc/group sur une machine Oracle Linux :

```
# cat /etc/group
root:x:0:
bin:x:1:
daemon:x:2:
...
wheel:x:10:student
...
student:x:1000:student
...
wagner:x:500:
#
```



Chaque ligne du fichier **/etc/group** correspond à un groupe et comporte 4 champs :

1. le **nom du groupe**,
2. le **mot de passe** du groupe qui est obsolète aujourd'hui,
3. le **GID** ("Group Id") du groupe,
4. la **liste des utilisateurs** qui sont membres du groupe.

Tout utilisateur peut à tout moment connaître son identité et son groupe à l'aide de la commande **id**.

Exemple 3.3 (id). Voici un exemple d'exécution de la commande id :

```
# id
uid=1000(student) gid=1000(student) groupes=1000(student),10(wheel) ...
```



Ici, l'utilisateur student a un groupe initial égal à 1000 (student) mais fait également partie du groupe 10 (wheel).



3.1.2 Les droits

Nous pouvons à présent distinguer

1. l'**utilisateur**,
2. le **groupe de l'utilisateur**,
3. les **autres**, c'est-à-dire les utilisateurs qui n'appartiennent pas au groupe de l'utilisateur.

Pour des raisons évidentes de sécurité, un utilisateur doit posséder des droits sur ce qui lui appartient mais n'est pas autorisé à manipuler les fichiers des autres utilisateurs sans autorisation. Il faut donc définir la notion de droits. On distingue les droits

1. en **lecture (r)**,
2. en **écriture (w)**,
3. en **exécution (x)**.

Lorsqu'un fichier (ou un répertoire) est créé, soit par commande, soit par un éditeur, soit par programmation, celui-ci appartient forcément à quelqu'un. Ce dernier a tous les droits sur ce fichier, on dit qu'il en est **propriétaire**. Il peut donc autoriser ou interdire l'accès à ce fichier (r, w, et/ou x), et cela par rapport à lui-même, par rapport aux utilisateurs du même groupe que lui, ou par rapport aux autres utilisateurs.

Exemple 3.4 (droits). Un fichier peut par exemple avoir les droits :

$\overbrace{r \ w \ x}$	$\overbrace{r \ - \ x}$	$\overbrace{\- \ - \ -}$
<i>Propriétaire</i>	<i>Groupe</i>	<i>Autres</i>

Dans ce cas,

- le propriétaire du fichier a les droits en lecture, écriture et exécution.
- les utilisateurs du même groupe que le propriétaire ont les droits en lecture et exécution sur le fichier, mais pas en écriture.
- les autres utilisateurs n'ont aucun droit sur le fichier.



Les droits d'un fichier sont représentés par **9 bits**. Si le droit est autorisé, cela correspond à un bit 1, sinon à un bit 0.

Exemple 3.5 (droits). Les droits du fichier de l'exemple 3.4 sont donc

$\overbrace{1 \ 1 \ 1}$	$\overbrace{1 \ 0 \ 1}$	$\overbrace{0 \ 0 \ 0}$
<i>Propriétaire</i>	<i>Groupe</i>	<i>Autres</i>

ce qui correspond en octal à 0750.



Les droits d'un répertoire

Dans le cas d'un répertoire, qui est un fichier particulier,

- le droit en **lecture** permet de **visionner le contenu** de ce répertoire,
- le droit en **exécution** permet de le **parcourir** à la recherche d'un nom de fichier, notamment à l'aide de la commande **cd**,

et cela pour le propriétaire, le groupe et les autres.

Exemple 3.6 (droits d'un répertoire). Dans l'exemple qui suit, les droits en lecture puis en exécution du répertoire AAA sont modifiés et cela pour son propriétaire :

```

# mkdir AAA
# ls -l
total 324
drwxr-xr-x. 2 student student 6 16 fév 15:14 AAA
...
...
# ls -l AAA
total 0
# touch AAA/Fich.dat
# cd AAA
# ls -l
total 0
-rw-r--r--. 1 student student 0 16 fév 15:17 Fich.dat
# cd ..
# chmod u-r AAA
# ls -l
total 324
d-wxr-xr-x. 2 student student 22 16 fév 15:17 AAA
...
...
# cd AAA
# ls -l
ls: impossible d'ouvrir le répertoire .: Permission non accordée
# cd ..
# ls -l AAA
ls: impossible d'ouvrir le répertoire AAA: Permission non accordée
# chmod u+r AAA
# chmod u-x AAA
# ls -l
total 324
drw-r-xr-x. 2 student student 6 16 fév 15:14 AAA
...
...
# ls -l AAA
ls: impossible d'accéder à AAA/Fich.dat: Permission non accordée
total 0
-???????????????? Fich.dat
# cd AAA
bash: cd: AAA: Permission non accordée
# chmod u+x AAA
#

```

Quelques remarques :

1. le caractère '**d**' apparaissant devant les droits correspond à "**directory**", il s'agit donc d'un répertoire.
2. la commande **chmod** permet d'attribuer un droit (+) ou retirer un droit (-) au propriétaire (u), au groupe (g) ou aux autres (o). Ce droit peut être en lecture (r), en écriture (w) ou en exécution (x).

Les répertoires des utilisateurs ont toujours les droits



tandis que les droits d'accès des répertoires traversés sont



ce qui permet de parcourir les répertoires, mais n'autorise pas d'écriture pour le groupe et pour les autres.

Exemple 3.7 (droits répertoire utilisateur). Voici un exemple d'exécution :

```
# cd /home
# ls -l
total 8
drwx----- . 39 student student 4096 12 fév 11:16 student
drwx----- . 15 wagner wagner 4096 11 fév 11:08 wagner
# cd wagner
bash: cd: wagner: Permission non accordée
# cd student
# ls -l
total 324
drwxr-xr-x . 2 student student 6 16 fév 15:14 AAA
...
...
#
#
```

Ici, l'utilisateur est student et n'a pas les droits pour accéder au répertoire de wagner. Le répertoire **/home** est un répertoire de la racine du disque contenant tous les répertoires utilisateurs.



3.1.3 Nom des fichiers

Un fichier est déterminé par son nom qui est composé d'une suite de caractères alphanumériques terminée par le caractères '**\0**'. Les différents répertoires traversés sont séparés par le caractère '**/**'.

La racine du disque est notée '**/**'. On distingue alors deux types de chemin pour accéder à un fichier :

1. **Chemin absolu** : Si le premier caractère est un '**/**', on doit alors indiquer tous les répertoires traversés à partir de la racine du disque.
2. **Chemin relatif** : Si le premier caractère n'est pas un '**/**', on partira du répertoire courant, c'est-à-dire celui où l'on se trouve.

Dans tout répertoire, il existe 2 répertoires particuliers :

1. **.** indique le **répertoire courant** (par défaut)
2. **..** indique le **répertoire parent**

Exemple 3.8. Utilisons les options de la commande ls :

```
# cd AAA
# ls -al
total 4
drwxr-xr-x. 2 student student 22 16 fév 15:17 .
drwxrwxr-x. 3 student student 4096 16 fév 15:16 ..
-rw-r--r--. 1 student student 0 16 fév 15:17 Fich.dat
#
```



Exemple 3.9. Voici quelques exemples de chemins absolus et relatifs :

Nom de fichier	Type de chemin	Explication
/home/AAA/Fich.dat	absolu	fichier Fich.dat qui se trouve dans le répertoire AAA qui se trouve dans le répertoire home qui se situe lui-même à la racine du disque
AAA/Fich.dat	relatif	fichier Fich.dat qui se trouve dans le répertoire AAA qui se trouve lui-même dans le répertoire courant
../Essai.dat	relatif	fichier Essai.dat qui se trouve dans le répertoire parent
./Fich.dat	relatif	fichier Fich.dat qui se trouve dans le répertoire courant



3.1.4 Les i-noeuds

Tout fichier sous Linux possède un i-noeud unique qui est une structure contenant toutes les informations propres à ce fichier. On parle d'i-noeud car les fichiers sont répartis sur disque dans des répertoires et des sous-répertoires partant de la racine du disque '/'. Le tout forme une arborescence dont

1. '/' est la racine,
2. les répertoires sont les noeuds intermédiaires,
3. les fichiers ordinaires (texte, code source, exécutable, ...) sont les noeuds finaux.

Tous les fichiers, y compris les répertoires, possèdent donc un **i-noeud unique**.

Un i-noeud contient les informations suivantes :

1. **UID et GID**, c'est-à-dire les identificateurs du propriétaire du fichier et du groupe dans lequel se trouve le propriétaire du fichier, on parle du groupe propriétaire.
2. **Type de fichier** : ordinaire, répertoire, pipe, socket, ...
3. **Droits du fichiers** en lecture/écriture/exécution pour le propriétaire, le groupe propriétaire et les autres. A noter que le "super-utilisateur root" possède tous les droits sur tous les fichiers du système.
4. **Taille du fichier** : le nombre de bytes contenu dans ce fichier. A savoir qu'il n'existe pas sous Linux de caractère de fin de fichier.

5. **Nombre de liens** : nombre de noms différents que possède ce fichier.
6. **3 dates** : date de dernier accès (en lecture), date de dernière modification du contenu du fichier, date de dernière modification de l'i-noeud.
7. **Table des adresses disque** des données du fichier : accès aux adresses physiques de tous les bytes du fichier sur disque.

Nous allons à présent montrer quelques exemples obtenus avec l'exécutable Test_stat décrit à l'**exemple 3.24**. Cet exécutable permet d'afficher le contenu de l'i-noeud d'un fichier.

Exemple 3.10 (création). Créons le fichier BBB.dat :

```
# touch BBB.dat
# date
mer fév 17 14:40:59 CET 2021
# ./Test_stat BBB.dat
User: 1000
Groupe: 1000
Droits d'accès: 0644
Type: fichier régulier
Taille: 0
Date dernier accès: Wed Feb 17 14:40:57 2021
Date dernière modification contenu: Wed Feb 17 14:40:57 2021
Date dernière modification inoeud: Wed Feb 17 14:40:57 2021
No inoeud: 205281963
#
```

La commande touch a créé le fichier BBB.dat de taille 0 en initialisant l'i-noeud avec les caractéristiques et les 3 dates du fichier. ■

Exemple 3.11 (lecture). Nous réalisons un accès en lecture au fichier BBB.dat :

```
# date
mer fév 17 14:49:01 CET 2021
# cat BBB.dat
# ./Test_stat BBB.dat
User: 1000
Groupe: 1000
Droits d'accès: 0644
Type: fichier régulier
Taille: 0
Date dernier accès: Wed Feb 17 14:49:05 2021
Date dernière modification contenu: Wed Feb 17 14:40:57 2021
Date dernière modification inoeud: Wed Feb 17 14:40:57 2021
No inoeud: 205281963
#
```

La lecture du contenu du fichier (vide ici) a simplement modifié la date de dernier accès au fichier. ■

Exemple 3.12 (Modification de l'i-noeud). Nous modifions l'i-noeud du fichier BBB.dat en modifiant ses droits :

```
# date
mer fév 17 14:55:37 CET 2021
# chmod 0600 BBB.dat
# ./Test_stat BBB.dat
User: 1000
Groupe: 1000
Droits d'accès: 0600
Type: fichier régulier
Taille: 0
Date dernier accès: Wed Feb 17 14:49:05 2021
Date dernière modification contenu: Wed Feb 17 14:40:57 2021
Date dernière modification inoeud: Wed Feb 17 14:55:47 2021
No inoeud: 205281963
#
```

La modification de l'i-noeud du fichier (les droits ici) a simplement modifié la date de dernière modification de l'i-noeud du fichier.

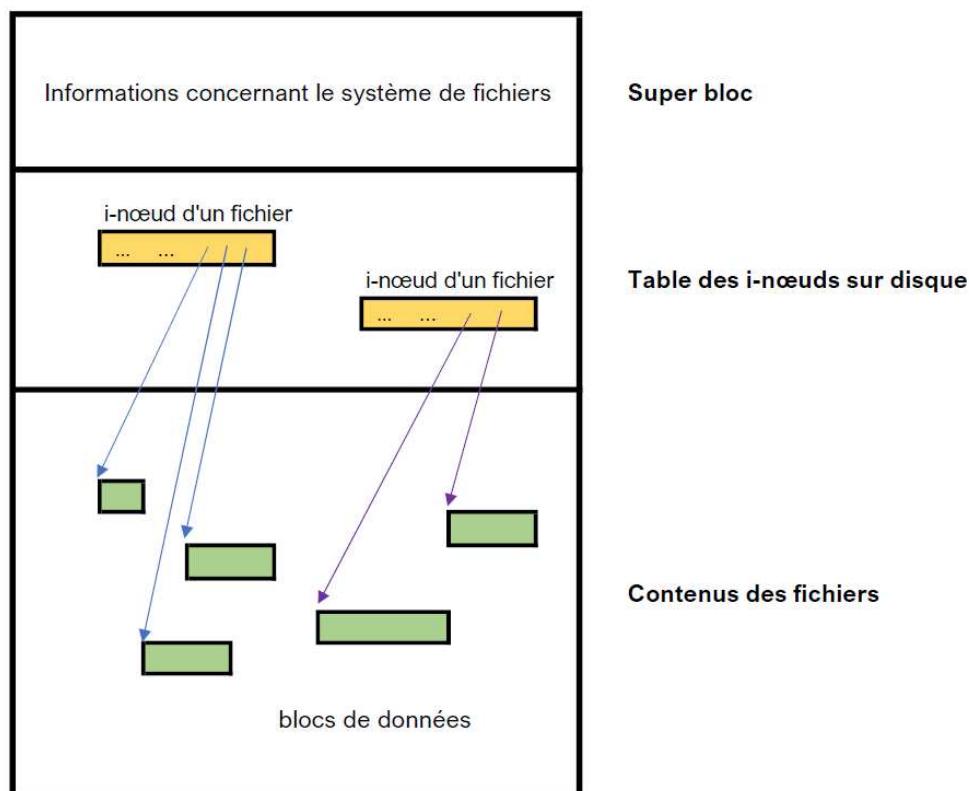


3.1.5 Structure d'un disque

D'une manière simplifiée, on peut dire qu'un disque est structuré selon 3 parties :

1. le **super bloc** contenant toutes les informations clés du système de fichiers,
2. la **table des i-noeuds** sur disque : elle correspond en quelque sorte à une table d'index de tous les fichiers du système de fichiers,
3. la zone des **blocs de données** de tous les fichiers sur disque. Un fichier peut ainsi comporter plusieurs blocs contigus ou non.

Les i-noeuds contiennent les adresses disques de tous les blocs de données d'un fichier et permettent ainsi de retrouver tous les "morceaux" d'un fichier afin de réaliser des lectures/écritures. Schématiquement, la structure d'un disque peut être représentée par



3.2 Appels bufferisés du système de fichiers

Les fonctions correspondantes sont normalement déjà bien connues et sont dites de "**haut niveau**". Elles manipulent un **buffer** dont le type est spécifié par **FILE***. Il s'agit des fonctions suivantes :

```
#include <stdio.h>

FILE* fopen(const char *path, const char *mode);
int fclose(FILE *fp);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fseek(FILE *stream, long offset, int whence);
int fflush(FILE *stream);
```

Ces fonctions travaillent dans l'espace adressable du processus, c'est-à-dire que le buffer associé au FILE* fait partie du processus et est donc une des variables du processus.

Lors d'une écriture (fwrite), le buffer n'est vidé et réellement écrit sur disque que dans les cas suivants :

- lorsqu'il est **plein** (difficilement prévisible),
- lors de l'appel à la fonction **fflush()**,
- lors de la fermeture du fichier par l'appel de **fclose()**,
- lors de la **fin normale** du processus (appel de la fonction `exit()`).

Dans les autres cas, il est impossible de prédire l'état du buffer. Ceci peut donner des résultats surprenant lorsque plusieurs processus manipulent le même fichier, ce qui est relativement fréquent sur une machine Linux.

Examinons deux exemples qui posent problème.

Exemple 3.13. Soit le programme suivant

```
/** ProblemeBuffer1.cpp ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Mercenier");
    // printf("Mercenier\n");
    write(1, "Denys", 5);
    exit(1);
}
```

dont l'exécution fournit

```
# ProblemeBuffer1
DenysMercenier#
```

La chaîne de caractères "Denys" s'affiche en premier lieu, malgré l'ordre d'appel des fonctions. En effet, en anticipant un peu, la fonction write est un appel non bufferisé et écrit directement sur le "canal 1" (équivalent non bufferisé du STDOUT). Le printf a écrit "Mercenier" dans le buffer STDOUT et celui-ci n'a pas été vidé avant l'appel de la fonction exit.

Remarquez que l'insertion du caractère '**\n**' dans un printf a non seulement pour effet le retour de ligne attendu mais également de vider le buffer STDOUT. En commentant le premier printf dans le programme précédent et en décommentant le second, l'exécution du programme fournit

```
# ProblemeBuffer1
Mercenier
Denys#
```



Exemple 3.14. Soit les deux programmes suivants

```
/** ProblemeBuffer2W.cpp ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    FILE *fb;

    if ((fb = fopen("Essai.txt","r+")) == NULL)
    {
        perror("(ProblemeBuffer2W) Err. de fopen");
        exit(1) ;
    }

    if ((fwrite("12345",5,1,fb)) != 1)
    {
        perror("(ProblemeBuffer2W) Err. de fwrite");
        exit(1) ;
    }

    // fflush(fb);
    sleep(5); // pour être certain que ProblemeBuffer2R ait eu
               // le temps de faire son fread

    if (fclose(fb))
    {
        perror("(ProblemeBuffer2W) Err. de fclose");
        exit(1) ;
    }
    exit(0);
}
```

et

```
/** ProblemeBuffer2R.cpp ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```

int main()
{
    FILE *fb;
    char buffer[80];

    if ((fb = fopen("Essai.txt","r+")) == NULL)
    {
        perror("(ProblemeBuffer2R) Err. de fopen");
        exit(1);
    }

    sleep(1); // pour etre certain que ProblemeBuffer2W ait eu le temps
               // de faire son fwrite

    if (fread(buffer,5,1,fb) != 1)
    {
        perror("(ProblemeBuffer2R) Err. de fread");
        exit(1);
    }

    printf("Lu : --%s--\n",buffer);

    if (fclose(fb))
    {
        perror("(ProblemeBuffer2R) Err. de fclose");
        exit(1);
    }

    exit(0);
}

```

et dont les exécutions fournissent

```

# cat Essai.txt
abcdefghijklmnopqrstuvwxyz
# ProblemeBuffer2W &
[1] 23962
# ProblemeBuffer2R
Lu : --abcde--
# cat Essai.txt
12345fghijklmnopqrstuvwxyz
[1]+ Fini ProblemeBuffer2W
#

```

Le processus ProblemeBuffer2W a écrit “12345” dans le fichier existant Essai.txt en utilisant la fonction fwrite, dont le buffer n'a pas été vidé immédiatement. Dès lors, au moment où le processus ProblemeBuffer2R lit dans le fichier, celui-ci n'a pas encore été modifié. Lorsque les deux processus sont terminés, le second appel à la commande cat montre que le fichier a pourtant été modifié. L'écriture a réellement été réalisée dans le fichier au moment où le processus ProblemeBuffer2W a exécuté la fonction fclose, c'est-à-dire bien après que le processus ProblemeBuffer2R ait lu dans le fichier.

Si maintenant on décommente, dans ProblemeBuffer2W.cpp, l'appel à **fflush(fb)**, l'exécution des deux programmes fournit alors

```
# cat Essai.txt
abcdefghijklmnopqrstuvwxyz
# ProblemeBuffer2W &
[1] 25731
# ProblemeBuffer2R
Lu : --12345--
# cat Essai.txt
12345fghijklmnopqrstuvwxyz
[1]+ Fini ProblemeBuffer2W
#
```

Dans ce cas-ci, le buffer fb de ProblemeBuffer2W a été vidé (et donc réellement écrit dans le fichier) avant que ProblemeBuffer2R ne lise dans le fichier. La lecture est donc “correcte”. ■

Ces deux exemples ont pour but de montrer que l’utilisation des appels bufferisés au système de fichiers peut poser des problèmes lorsque plusieurs processus manipulent simultanément le même fichier. Voilà pourquoi, il est nettement plus courant, dans le monde Linux, d’utiliser les appels non bufferisés au système de fichiers. Ceux-ci sont abordés à la section suivante.

3.3 Appels système du système de fichiers

Il existe des fonctions de gestion de fichiers similaires aux fonctions de la section précédente mais n’utilisant pas de buffer. Ces fonctions sont dites de “**bas niveau**” et sont plutôt appellées “**appels système**” ou des “**primitives**” car elles manipulent directement des tables du système d’exploitation. Néanmoins, l’accès aux fichiers se fera toujours d’une manière similaire :

- Ouverture/création : **open()**, **creat()**
- Lecture/écriture : **read()**, **write()**
- Accès direct à un endroit du fichier : **lseek()**
- Fermeture d’un fichier : **close()**

D’autres fonctions spécifiques à Linux seront également abordées :

- Changement de propriétaire et droits : **chown()**, **chmod()**
- Lecture/modification des attributs d’un fichier : **stat()**
- Création/suppression de liens symboliques : **link()**, **unlink()**
- Modification du mode d’ouverture/pose de verrous : **fcntl()**

3.3.1 Tables gérées par le système

Tous ces appels système ne manipulent pas de buffer, donc la structure FILE* n’est plus utilisée ici. Pour accéder à un fichier, un processus ne peut donc plus utiliser son buffer associé. La notion est ici remplacée par celle de **descripteur de fichier**. Un descripteur de fichier est

un **entier positif** connu du processus et qui lui permet d'accéder au fichier associé.

En parallèle de ces descripteurs de fichiers, le système gère deux tables **communes** à tous les processus :

1. la **table des fichiers ouverts**,
2. la **table des i-noeuds en mémoire**.

Ceci est illustré à la figure 3.1. Chaque ligne de chacune de ces 2 tables est appelée "**une entrée**" dans cette table et chaque entrée comporte plusieurs informations (les colonnes), soit sur le mode d'ouverture (table des fichiers ouverts), soit sur le fichier lui-même (table des i-noeuds en mémoire).

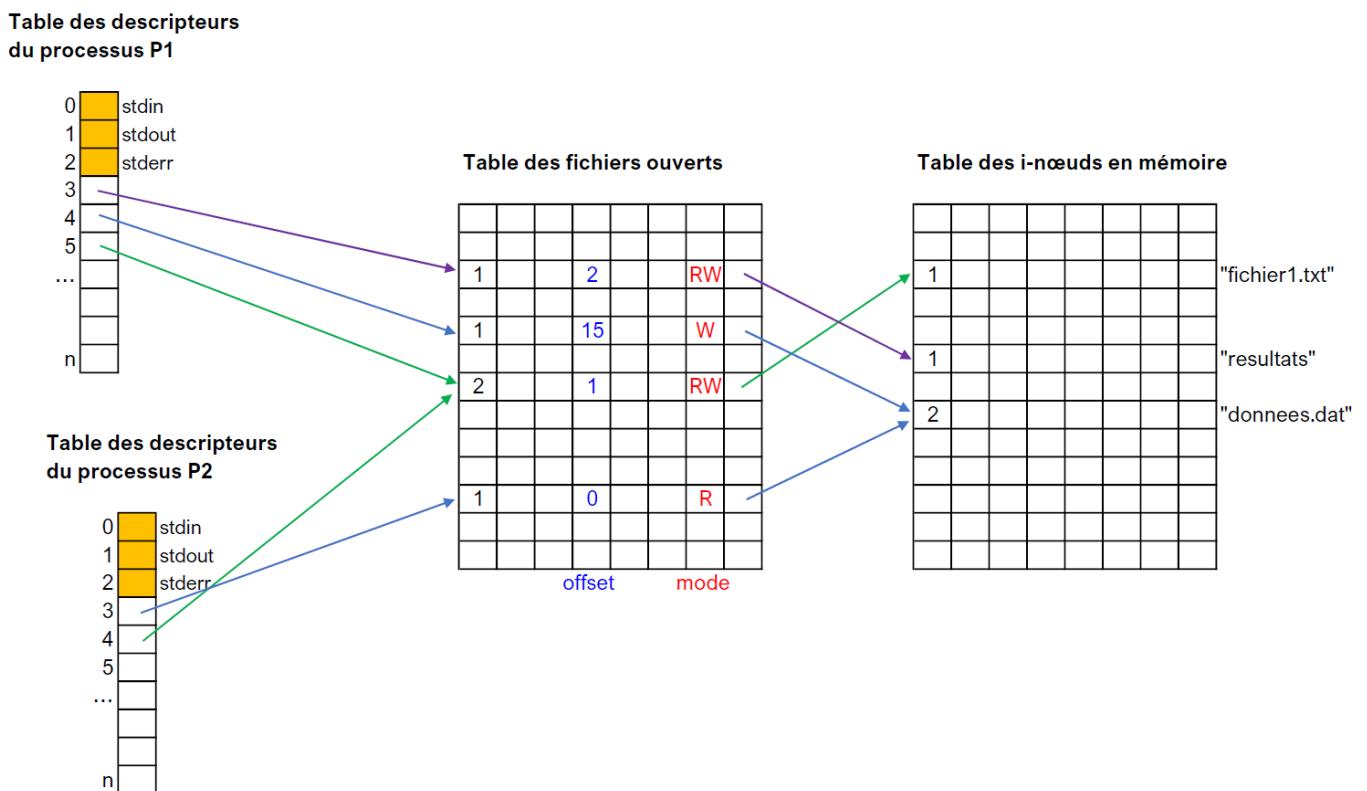


FIGURE 3.1 – Tables système de gestion des fichiers.

Table des descripteurs

Cette table est **propre** aux processus, c'est-à-dire que chaque processus possède sa propre table de descripteurs.

Chaque fois qu'un fichier est ouvert, une entrée dans cette table est attribuée par le système (soit par la primitive open, soit par les primitives dup et dup2 ; voir plus loin).

Les 3 premières entrées sont réservées pour

1. l'entrée standard : **0 stdin**
2. la sortie standard : **1 stdout**
3. la sortie d'erreur : **2 stderr**

Chaque entrée dans cette table pointe sur une entrée dans la table des fichiers ouverts.

Table des fichiers ouverts

Cette table est **commune** à tous les processus et est donc unique. Elle contient les informations relatives à l'**ouverture des fichiers**. Chaque entrée comporte notamment :

- un **offset** c'est-à-dire la position actuelle dans le fichier (ou pointeur de fichier ou tête de "lecture et/ou écriture"). Lors de l'ouverture d'un fichier, celui-ci est initialisé à 0 (sauf si ouverture avec écriture en fin de fichier).
- le **mode d'ouverture** : lecture seule, écriture seule, lecture/écriture, ...
- un **compteur** (première colonne dans la table de la figure 3.1) représentant le nombre de descripteurs pointant sur cette entrée.

Table des i-noeuds en mémoire

Cette table est également **commune** à tous les processus et est unique. Elle contient les informations relatives aux fichiers. Chaque entrée dans cette table

- correspond à une copie en mémoire de l'**i-noeud** du fichier ouvert correspondant.
- comporte un champ (première colonne de la table de la figure 3.1) correspondant à un **compteur** représentant le nombre d'entrées dans la table des fichiers pointant vers elle.

Remarque importante

Un fichier physique (c'est-à-dire présent sur disque) ouvert par un ou plusieurs processus ne présente qu'**une et une seule entrée dans la table des i-noeuds en mémoire**. On peut dire également que cette table représente l'**ouverture "physique"** d'un fichier.

Par contre, ce même fichier peut être ouvert plusieurs fois et de manières différentes (mode d'ouverture, offset, ...). Ceci est représenté par plusieurs entrées distinctes dans la table des fichiers ouverts et pointant vers l'i-noeud unique en mémoire de ce fichier. On pourrait alors dire que la table des fichiers ouverts représente l'**ouverture "logique"** d'un fichier.

Exemple 3.15. Reprenons les différents exemples de la figure 3.1 :

- Le fichier "**resultat**" est ouvert physiquement, et logiquement une seule fois par le processus P1 qui peut le manipuler avec son descripteur 3. Via ce descripteur, le fichier est ouvert en lecture/écriture et la position de la lecture/écriture (offset) est de 2 bytes par rapport au début du fichier.
- Le fichier "**donnees.dat**" est ouvert physiquement, et logiquement deux fois :
 1. par le processus P1 qui peut le manipuler avec son descripteur 4. Via ce descripteur, le fichier est ouvert en écriture seule et sa tête d'écriture (offset) est de 15 bytes par rapport au début du fichier.
 2. par le processus P2 qui peut le manipuler avec son descripteur 3. Via ce descripteur, le fichier est ouvert en lecture seule et sa tête de lecture (offset) est au début du fichier.

- le fichier “**fichier1.txt**” est ouvert physiquement et logiquement une seule fois, par le processus P1 (qui le manipule avec son descripteur 5) et par le processus P2 (qui le manipule avec son descripteur 4). Les deux processus pointent donc vers la même entrée la table des fichiers ouverts. Ils ont donc le même offset et toute manipulation de cet offset (par les appels à read, write ou lseek) par un des processus affecte l'autre processus.



Nous allons à présent étudier les différentes primitives permettant de manipuler un fichier et donc les tables système relatives aux fichiers.

3.3.2 La primitive “open”

Cette fonction permet de créer un fichier et/ou de l'ouvrir avec le mode d'ouverture désiré. Toute la documentation peut être obtenue dans le man :

```
# man -s 2 open
...
#
```

Cette fonction existe en 2 versions :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Paramètres

Le paramètre **mode** n'est utilisé que lorsqu'on désire créer un nouveau fichier. Il s'agit des droits (lecture/écriture/exécution pour le propriétaire, le groupe et les autres) donnés au fichier lors de sa création (voir plus loin).

Dans tous les cas, voici la signification des autres paramètres :

Paramètre	Signification
pathname	nom du fichier
flags	mode d'ouverture (lecture seule, écriture seule, lecture/écriture, ...)

Toutes les valeurs possibles prises par le paramètre **flags** sont définies dans le fichier **fcntl.h** et les principales sont reprises ici :

flags	Signification
O_RDONLY	Lecture seule
O_WRONLY	Ecriture seule
O_RDWR	Lecture/écriture
O_CREAT	Si le fichier n'existe pas, il est créé. Dans ce cas, le paramètre mode détermine les droits du fichier sur disque.
O_EXCL	Provoque une erreur (retour -1) si le fichier existe déjà et que le flag O_CREAT est positionné également. Cette option évite d'effacer un fichier qui existe déjà.
O_TRUNC	Détruit le contenu du fichier s'il existe déjà
O_APPEND	Ecriture en fin de fichier
O_SYNC	Mode synchrone. Toute écriture (appel à write) est bloquante jusqu'à ce que l'écriture ait réellement été réalisée sur disque.

Plusieurs valeurs peuvent être utilisées simultanément en les combinant par un OU logique bit à bit, c'est-à-dire en utilisant l'opérateur | (voir plus loin).

Algorithme

En fonction des paramètres reçus, l'appel système open manipule les tables système de la manière suivante :

Algorithme 3.1 La primitive ou l'appel système open.

```

Si i-noeud du fichier non présent dans la table des i-noeuds en mémoire
    Alors
        Si fichier n'existe pas sur disque
            Alors // on tente de le créer
                Si flags contient O_CREAT
                    Alors le fichier est créé sur disque
                    Sinon retour -1 et errno positionné à ENOENT
                Fin Si
            Fin Si
        i-noeud du fichier chargé dans la table des i-noeuds en mémoire
    Fin Si
Si accès autorisé // le processus a les droits spécifiés par flags sur le fichier
    Alors Ajout nouvelle entrée dans table des fichiers ouverts et retour valeur
        du descripteur.
    Sinon retour -1 et errno positionné à EACCES
Fin Si

```

Pour vérifier si le processus est autorisé à accéder au fichier, la fonction open utilise l'i-noeud en mémoire pour comparer les droits du fichier (lecture/écriture/exécution pour le propriétaire, le groupe et les autres) et le mode d'ouverture souhaité (lecture seule, écriture seule, ...) par le processus. Pour que l'accès soit autorisé, il faut que le propriétaire du processus ait les droits suffisants sur le fichier.

Valeurs de retour

Le retour de la fonction est

- la **valeur du descripteur** attribuée par le système, il s'agit de la **première entrée libre** dans la table des descripteurs du processus, ou

- 1 en cas d'erreur et, errno est positionné.

Si on ne fait rien de particulier pour changer cela, les 3 premières entrées (0, 1, 2) dans la table des descripteurs du processus sont réservées. Dans ce cas, la valeur de retour de la primitive open est un entier supérieur ou égal à 3.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	L'accès demandé au fichier est interdit
EEXIST	Le fichier existe et les flags O_CREAT O_EXCL sont spécifiés
EFAULT	L'adresse du nom du fichier est invalide
EMFILE	Le nombre maximal de fichiers ouverts par le processus est atteint
ENFILE	Le nombre maximal de fichiers ouverts par le système est atteint
ENOENT	Le fichier n'existe pas et le flag O_CREAT n'est pas spécifié
EINTR	L'appel système a été interrompu par un signal
...	...

3.3.3 La primitive “close”

Cette fonction permet de fermer un fichier dont on connaît le descripteur. La documentation peut être obtenue dans le man :

```
# man -s 2 close
...
#
```

Le prototype de cette fonction est :

```
#include <unistd.h>

int close(int fd);
```

Paramètre

Le paramètre **fd** correspond simplement au descripteur du fichier que l'on désire fermer.

Algorithme

Tout comme la primitive open, la primitive close manipule les tables système de la manière suivante :

Algorithme 3.2 La primitive ou l'appel système close.

Le compteur de l'entrée dans la table des fichiers ouverts pointée par le descripteur fd est décrémenté de 1 et l'entrée correspondante dans la table des descripteurs du processus est libérée.

Si ce compteur est égal à 0

Alors

Cette entrée dans la table des fichiers ouverts est supprimée.

Le compteur de l'entrée dans la table des i-noeuds en mémoire pointée par cette entrée est décrémenté de 1.

Si ce compteur est égal à 0

Alors l'entrée dans la table des i-noeuds en mémoire est supprimée

Fin Si

Fin Si

Si plus aucun descripteur ne pointe vers une entrée dans la table des fichiers ouverts, son compteur est à zéro, elle ne sert plus à rien et est supprimée de la table. De la même manière, si plus aucune entrée de la table des fichiers ouverts ne pointe sur un i-noeud dans la table des i-noeuds en mémoire, son compteur est à zéro, il ne sert plus à rien et est supprimé de la table.

Une fois un descripteur libéré par la fonction close, le numéro correspondant pourra être réutilisé par la suite.

Valeurs de retour

Le retour de la fonction est

- **0** si le fichier a pu être fermé correctement, ou
- **-1** en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs de errno sont

errno	signification
EBADF	Le descripteur fd n'est pas valide
EINTR	L'appel système a été interrompu par un signal
EIO	Une erreur d'entrée/sortie s'est produite.

Exemple 3.16 (open/close). Soit le programme suivant

```
***** Test_OpenClose.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fd;

    printf("pid = %d\n",getpid());

    if ((fd = open("Essai.dat",O_RDWR|O_CREAT|O_EXCL,0644)) == -1)
    {
```

```

    perror("Erreur de open()");
    exit(1);
}

sleep(10); // pour pouvoir faire un lsof

if (close(fd))
{
    perror("Erreur de close()");
    exit(1);
}

exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./Test_OpenClose &
[1] 9899
pid = 9899
# lsof -p 9899 -a -d "0-10"
lsof: WARNING: can't stat() tracefs file system /sys/kernel/debug/tracing
Output information may be incomplete.
COMMAND   PID   USER FD   TYPE DEVICE SIZE/OFF      NODE NAME
Test_Open 9899 student  0u   CHR  136,1        0t0          4 /dev/pts/1
Test_Open 9899 student  1u   CHR  136,1        0t0          4 /dev/pts/1
Test_Open 9899 student  2u   CHR  136,1        0t0          4 /dev/pts/1
Test_Open 9899 student  3u   REG  252,0        0 201652862 /home/student/Essai.dat
# ./Test_OpenClose
pid = 9935
Erreur de open(): File exists
[1]+  Fini ./Test_OpenClose
# ls -l Essai.dat
-rw-r--r--. 1 student student 0 22 jan 14:15 Essai.dat
#

```

Dans cet exemple :

- Il s'agit d'une ouverture avec création. Les droits du fichiers sur disque sont 644 en octal, c'est-à-dire 110100100 en binaire, ou encore `rw-r--r--`.
- La fonction `open` a retourné le descripteur 3, bien visible dans la table des descripteurs du processus. Une fois ouvert via ce descripteur, le fichier est ouvert en lecture/écriture.
- Lors du 2ème lancement du programme, la fonction `open` a retourné -1 et le processus s'est arrêté. En effet, le flag `O_EXCL` étant positionné et le fichier "Essai.dat" existant déjà, celui-ci n'a pas été écrasé.



3.3.4 Les primitives "write" et "read"

Ces fonctions permettent d'écrire et de lire dans un fichier dont on connaît le descripteur. La documentation peut être obtenue dans le man :

```
# man -s 2 write
...
# man -s 2 read
...
#
```

Les prototypes de ces fonctions sont :

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

Paramètres

Les paramètres de ces deux fonctions sont

Paramètre	Signification
fd	descripteur du fichier
buf	cas du write : adresse de la variable ou du “paquet de bytes” que l’on désire écrire sur disque. cas du read : adresse de la variable ou de la zone mémoire destinée à recevoir les bytes qui vont être lus.
count	nombre de bytes que l’on désire écrire/lire.

Valeurs de retour

Le retour de la fonction **write** est

- égal à **count**, ce qui signifie que l’écriture s’est bien déroulée. Dans ce cas, la fonction retourne le nombre de bytes qui ont été écrits sur disque et ce nombre correspond aux nombre de bytes que l’on a voulu écrire.
- **-1** en cas d’erreur et, errno est positionné.

Le retour de la fonction **read** est

- égal à un **nombre >= 0**, ce qui signifie que la lecture s’est bien déroulée. Dans ce cas, la fonction retourne le nombre de bytes qui ont été lus sur disque.
- **-1** en cas d’erreur et, errno est positionné.

En fait, sous Linux, il n’existe **pas de caractère de fin de fichier**. Pratiquement, lors d’une lecture, la fin de fichier est détectée et atteinte lorsque le nombre bytes lus (retour de la fonction read) ne correspond pas au nombre de bytes que l’on voulait lire (paramètre count).

En cas d’erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EBADF	Le descripteur fd n'est pas valide
EFAULT	L'adresse buf n'est pas valide.
EINTR	L'appel système a été interrompu par un signal
EIO	Une erreur d'entrée/sortie s'est produite.
...	...

3.3.5 La primitive “lseek”

Cette fonction permet de déplacer la tête de lecture et/ou d'écriture dans un fichier dont on connaît le descripteur. La documentation peut être obtenue dans le man :

```
# man -s 2 lseek
...
#
```

Le prototype de cette fonction est :

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

off_t lseek(int fd, off_t offset, int whence);
```

Paramètres

Les paramètres de cette fonction sont

Paramètre	Signification
fd	descripteur du fichier
offset	nb de bytes souhaités pour le déplacement. Cette valeur est un nombre entier qui peut être positif, négatif ou nul .
whence	est un entier permettant de spécifier l'origine du déplacement

Le paramètre whence peut prendre une des valeurs (MACRO) suivantes :

whence	Signification
SEEK_SET	La tête est placée à offset bytes par rapport au début du fichier .
SEEK_CUR	La tête est avancée de offset bytes par rapport à la position actuelle .
SEEK_END	La tête est placée à offset bytes par rapport à la fin du fichier .

Le début du fichier correspond à la valeur 0 de l'offset dans la table des fichiers ouverts. Tout déplacement par un lseek sera autorisé à condition que la position finale soit une valeur entière positive ou nulle. Il est donc **interdit de se placer “avant” le début du fichier**.

Valeurs de retour

Le retour de la fonction **lseek** est

- égal à un **nombre ≥ 0** , qui correspond à la position dans le fichier après déplacement de la tête de lecture/écriture.

- 1 en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EBADF	Le descripteur fd n'est pas valide
ESPIPE	Le descripteur fd est associé à un pipe (voir chapitre ultérieur).
EINVAL	Soit whence n'est pas valable, soit la position finale demandée dans le fichier est négative.
...	...

Exemple 3.17 (lseek). Soit le programme suivant

```
***** Test_ReadWriteLseek.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fd, rc;
    char buffer[20];

    printf("pid = %d\n",getpid());

    if ((fd = open("Essai.txt",O_RDWR)) == -1) {
        perror("Erreur de open()");
        exit(1);
    }

    // Positionnement en fin de fichier
    if ((rc = lseek(fd,0,SEEK_END)) == -1) perror("Erreur de lseek(1)");
    printf("Taille du fichier : %d\n",rc);

    // Positionnement en offset = 5
    if ((rc = lseek(fd,5,SEEK_SET)) == -1) perror("Erreur de lseek(2)");
    printf("Position dans le fichier : %d\n",rc);

    // On écrit 6 bytes
    if (write(fd,"WAGNER",6) != 6) perror("Erreur de write(1)");

    // Positionnement dans le fichier ?
    if ((rc = lseek(fd,0,SEEK_CUR)) == -1) perror("Erreur de lseek(3)");
    printf("Position dans le fichier : %d\n",rc);

    // On recule de 20 bytes
    if ((rc = lseek(fd,-20,SEEK_CUR)) == -1) perror("Erreur de lseek(4)");
    printf("Position dans le fichier : %d\n",rc);

    // On lit 5 bytes
    if ((rc = read(fd,buffer,5)) == -1) perror("Erreur de read(1)");
    buffer[rc] = '\0'; // sinon buffer ne contient pas une chaîne de caractères
    printf("Lu (%d/5) : --%s--\n",rc,buffer);

    // On lit 15 bytes
    if ((rc = read(fd,buffer,15)) == -1) perror("Erreur de read(2)");
    buffer[rc] = '\0';
```

```

printf("Lu (%d/15) : --%s--\n",rc,buffer);

// Positionnement en offset = 30
if ((rc = lseek(fd,30,SEEK_SET)) == -1) perror("Erreur de lseek(5)");
printf("Position dans le fichier : %d\n",rc);

// On lit 5 bytes
if ((rc = read(fd,buffer,5)) == -1) perror("Erreur de read(3)");
buffer[rc] = '\0';
printf("Lu (%d/5) : --%s--\n",rc,buffer);

sleep(10); // pour faire un lsof

if (close(fd)) {
    perror("Erreur de close()");
    exit(1);
}

exit(0);
}

```

dont un exemple d'exécution fournit

```

# cat Essai.txt
abcdefghijklmnopqrstuvwxyz
# ./Test_ReadWriteLseek &
[1] 16874
pid = 16874
Taille du fichier : 27
Position dans le fichier : 5
Position dans le fichier : 11
Erreur de lseek(4): Invalid argument
Position dans le fichier : -1
Lu (5/5) : --lmnop--
Lu (11/15) : --qrstuvwxyz
--
Position dans le fichier : 30
Lu (0/5) : ----
# lsof -p 16874 -aod "0-10"
lsof: WARNING: can't stat() tracefs file system /sys/kernel/debug/tracing Output
information may be incomplete.
COMMAND      PID      USER FD   TYPE DEVICE OFFSET      NODE NAME
Test_Read 16874 student  0u  CHR  136,1    0t0          4 /dev/pts/1
Test_Read 16874 student  1u  CHR  136,1    0t0          4 /dev/pts/1
Test_Read 16874 student  2u  CHR  136,1    0t0          4 /dev/pts/1
Test_Read 16874 student  3u  REG  252,0  0t30 206807623 /home/student/Essai.txt
# cat Essai.txt
abcdeWAGNERlmnopqrstuvwxyz
[1]+ Fini ./Test_ReadWriteLseek
#

```

Dans cet exemple :

- Le premier positionnement (lseek(1)) a permis de déterminer la taille du fichier.
- Le 4ème positionnement (lseek(4)) a provoqué un erreur car la position finale dans le fichier

aurait du correspondre à la valeur -9, ce qui n'est pas autorisé.

- Lors de la seconde lecture (read(2), le nombre de nombre de bytes demandés est 15 mais il ne reste que 11 bytes dans le fichier, la fonction read lit donc les 11 bytes restants et retourne la valeur 11. Le retour de ligne après le 'z' correspond au fait que le caractère '\n' a été lu dans le fichier.
- Le 5ème positionnement (lseek(5)) est autorisé même si on sort du fichier.
- Bien que l'on soit hors du fichier (position actuelle = 30), la fonction read (3) ne retourne pas -1 mais bien 0 car la fin du fichier a déjà été atteinte. Il ne s'agit pas d'un cas d'erreur.



3.3.6 La primitive “chmod”

Cette fonction permet de modifier les **droits en lecture/écriture/exécution** d'un fichier sur disque. Elle modifie donc l'i-noeud du fichier et non son contenu. La documentation peut être obtenue dans le man :

```
# man -s 2 chmod
...
#
```

Le prototype de cette fonction est :

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

int chmod(const char* path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Cette fonction existe en deux versions mais celles-ci font exactement la même chose. La première reçoit le nom du fichier à modifier tandis que la seconde reçoit le descripteur du fichier à modifier si celui-ci a été ouvert précédemment.

Paramètres

Les paramètres de cette fonction sont

Paramètre	Signification
fd	descripteur du fichier
path	nom du fichier
mode	les nouveaux droits

Valeurs de retour

Le retour de la fonction **chmod** ou **fchmod** est

- **0** si la modification a pu être réalisée.
- **-1** en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EROFS	Le fichier spécifié réside sur un système de fichiers en lecture seule.
EPERM	Le processus n'a pas les droits suffisants sur le fichier.
EBADF	Le descripteur n'est pas valide.
...	...

Exemple 3.18 (chmod). Soit le programme suivant

```
***** Test_chmod.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main()
{
    if (chmod("Essai.dat",0777))
    {
        perror("Erreur de chmod()");
        exit(1);
    }
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -l Essai.dat
-rw-r--r--. 1 student student 0 5 fév 08:11 Essai.dat
# ./Test_chmod
# ls -l Essai.dat
-rwxrwxrwx. 1 student student 0 5 fév 08:11 Essai.dat
```



3.3.7 La primitive “access”

Cette fonction permet au **processus** appelant de **tester ses droits en lecture/écriture/exécution** sur un fichier. La documentation peut être obtenue dans le man :

```
# man -s 2 access
...
#
```

Le prototype de cette fonction est :

```
#include <unistd.h>

int access(const char* pathname, int mode);
```

Paramètres

Les paramètres de cette fonction sont

Paramètre	Signification
pathname	le nom du fichier à tester
mode	est un entier permettant de spécifier le droit à tester

Le paramètre mode peut prendre une des valeurs (MACRO) suivantes :

whence	Signification
F_OK	Permet de tester l' existence du fichier.
R_OK	Permet de tester l'accès en lecture au fichier
W_OK	Permet de tester l'accès en écriture au fichier
X_OK	Permet de tester l'accès en exécution au fichier

Valeurs de retour

Le retour de la fonction **access** est

- **0** si le processus possède le droit demandé.
- **-1** sinon ou en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EACCESS	L'accès au fichier lui-même est refusé.
EFAULT	pathname ne pointe pas sur une chaîne de caractères valide.
...	...

Exemple 3.19 (access). Soit le programme suivant

```
***** Test_access.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (!access(argv[1],X_OK))
        printf("Le processus a les droits en execution sur le fichier.\n");
    else perror("Erreur de access()");

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_access Test_access.cpp
Erreur de access(): Permission denied
# ./Test_access Test_access
Le processus a les droits en execution sur le fichier.
#
```

Dans le premier cas, le fichier testé est un fichier source, donc non exécutable, tandis que dans le second cas, le processus teste son propre fichier exécutable.

3.3.8 La primitive “umask”

Lorsqu'un processus crée un nouveau fichier, le système gère un **mécanisme de sécurité** qui vise à protéger l'utilisateur dans le sens où les droits (en lecture/écriture/exécution) du fichier créé sur disque ne correspondent pas strictement aux droits demandés dans la fonction open. Les droits obtenus seront alors **plus restrictifs** que ceux demandés.

Pour cela, le fichier **/etc/profile** contient une variable **umask** :

```
# cat /etc/profile  
...  
umask 022  
...  
#
```

Cette variable (022 en octal ou 000 010 010 en binaire) est combinée avec les droits demandés afin d'obtenir les droits qui seront effectivement obtenus, et cela de la manière suivante :

droits obtenus = droits souhaités & ~umask

Exemple 3.20. Si nous souhaitons les droits `rwxrwxrwx` (777) pour un fichier, nous obtenons réellement les droits suivants

droits souhaités	=	0777	=	111 111 111
& ~umask	=	0755	=	111 101 101
droits obtenus	=	0755	=	111 101 101

Les droits réellement obtenus sont donc `rwxr-xr-x`. Vérifions cela à l'aide du programme suivant :

```
***** Test_umask.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

    if ((fd = open(argv[1],O_RDWR | O_CREAT | O_EXCL, 0777)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_umask data.dat
# ls -l data.dat
-rwxr-xr-x. 1 student student 0 11 fév 11:12 data.dat
#
```

Si on veut éviter ce mécanisme de sécurité ou modifier la valeur de la variable umask, plusieurs solutions existent :

- Modifier directement le fichier **/etc/profile**. Dans ce cas, il faut disposer des droits de root et la nouvelle valeur de umask sera appliquée à **tous les fichiers créés par tous les utilisateurs**.
 - Modifier le fichier **.bashrc** situé dans le répertoire de l'utilisateur. Dans ce cas, la valeur de la variable umask redéfinie dans ce fichier sera appliquée à **tous les fichiers créés par l'utilisateur**.
 - Utiliser la **fonction umask()** décrite ci-dessous.

La fonction umask permet de **modifier la valeur de la variable umask localement**, c'est-à-dire au sein même du processus. La documentation peut être obtenue dans le man :

```
# man -s 2 umask  
...  
#
```

Le prototype de cette fonction est :

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

mode_t umask(mode_t mask);
```

Paramètre

Le paramètre de cette fonction est

Paramètre	Signification
mask	la nouvelle valeur d'umask à appliquer

Valeur de retour

Cette fonction n'échoue jamais, la valeur de retour est toujours l'**ancienne valeur de la variable umask**.

Exemple 3.21 (umask). Nous allons fixer la valeur de l'umask à 077 (~700) afin de ne pas altérer les droits demandés pour l'utilisateur mais supprimer tous ceux du groupe et des autres, ce qui correspond à

droits souhaités	=	0777	=	111 111 111
& ~umask	=	0700	=	111 000 000
droits obtenus	=	0700	=	111 000 000

Donc, soit le programme suivant

```
***** Test_umask.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc,char* argv[])
{
    int oldMask;
    int fd;

    oldMask = umask(~0700);
    printf("oldMask = %o\n",oldMask);

    if ((fd = open(argv[1],O_RDWR | O_CREAT | O_EXCL, 0777)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_umask data.dat
oldMask = 022
# ls -l data.dat
-rwx----- . 1 student student 0 11 fév 11:15 data.dat
#
```

Nous observons bien que l'ancienne valeur de umask est 022 et que les droits réellement obtenus sont rwx-----.



3.3.9 Les primitives “dup” et “dup2”

Ces fonctions permettent de **dupliquer un descripteur de fichier ouvert**. Il sera alors possible au processus d'accéder au même fichier indépendamment par l'un ou l'autre des descripteurs. Ces deux descripteurs pointent donc sur la **même entrée dans la table des fichiers ouverts**.

Une des utilités de ces fonctions est de pouvoir **rediriger les entrées/sorties standards** (stdin, stdout, stderr ; descripteurs 0, 1 et 2) vers des fichiers ou vers les entrées/sorties d'un pipe de communication, comme cela sera abordé plus tard dans ce cours.

La documentation peut être obtenue dans le man :

```
# man -s 2 dup
...
#
```

Les prototypes de ces fonctions sont :

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Paramètres

Les paramètres de ces fonctions sont

Paramètre	Signification
oldfd	le numéro du descripteur à dupliquer
newfd	le numéro du descripteur dupliqué

La fonction **dup** recherche la **plus petite entrée libre dans la table descripteurs** du processus et lui attribue le descripteur dupliqué.

La fonction **dup2** transforme newfd en une copie de oldfd, en **fermant** (close) automatiquement **newfd**, si besoin est, mais :

- Si oldfd n'est pas un descripteur de fichier valable, alors l'appel échoue et newfd n'est pas fermé.
- Si oldfd est un descripteur de fichier valable et newfd a la même valeur que oldfd, alors l'appel ne fait rien et renvoie newfd.

Lorsque l'appel de la fonction réussit, le **compteur** de l'entrée correspondante dans la table des fichiers ouverts est **incrémenté de 1**.

Valeurs de retour

Le retour de ces fonctions est

- **le nouveau descripteur** en cas de succès.
- **-1** en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EBADF	oldfd n'est pas un descripteur de fichier ouvert, ou newfd n'est pas dans les valeurs autorisées pour un descripteur.
EINTR	l'appel système a été interrompu par un signal.
EMFILE	le processus dispose déjà du nombre maximal de descripteurs de fichier autorisés simultanément.
...	...

Exemple 3.22 (dup). Soit le programme suivant

```
***** Test_dup.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    int fd;
    int fdDup;

    printf("pid : %d\n",getpid());

    if ((fd = open("Essai.txt",O_RDWR)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    if ((fdDup = dup(fd)) == -1)
    {
        perror("Erreur de dup()");
        exit(1);
    }

    if (lseek(fd,0,SEEK_END) == -1)
    {
        perror("Erreur de lseek()");
        exit(1);
    }

    sleep(10);
    printf("\nSuite\n");

    if (write(fdDup,"WAGNER\n",7) == -1)
    {
        perror("Erreur de write()");
        exit(1);
    }

    sleep(10);
    exit(0);
}
```

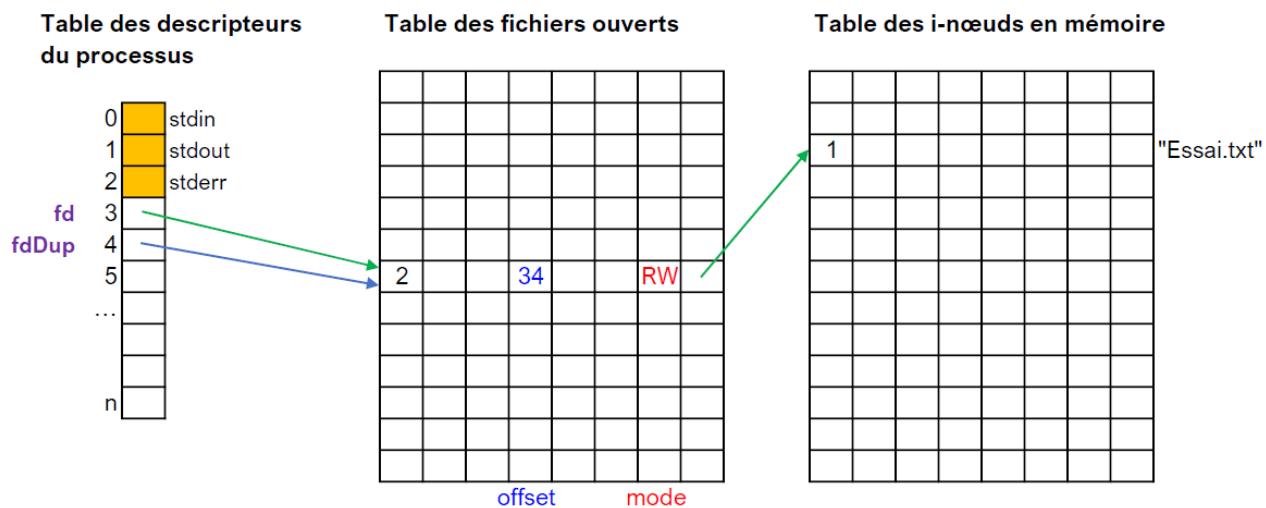
dont un exemple d'exécution fournit

```
# cat Essai.txt
abcdefghijklmnopqrstuvwxyz
# ./Test_dup &
[1] 5027
pid : 5027
# lsof -p 5027 -aod "0-10"
COMMAND PID USER FD TYPE DEVICE OFFSET NODE NAME
Test_dup 5027 student 0u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 1u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 2u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 3u REG 252,0 0t27 205281955 /home/student/Essai.txt
Test_dup 5027 student 4u REG 252,0 0t27 205281955 /home/student/Essai.txt
#
Suite
# lsof -p 5027 -aod "0-10"
COMMAND PID USER FD TYPE DEVICE OFFSET NODE NAME
Test_dup 5027 student 0u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 1u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 2u CHR 136,1 0t0 4 /dev/pts/1
Test_dup 5027 student 3u REG 252,0 0t34 205281955 /home/student/Essai.txt
Test_dup 5027 student 4u REG 252,0 0t34 205281955 /home/student/Essai.txt
# cat Essai.txt
abcdefghijklmnopqrstuvwxyz
WAGNER
[1]+ Fini ./Test_dup
#
```

Nous observons bien que

- L'ouverture du fichier a fourni le descripteur **fd=3**.
- La duplication du descripteur fd par la fonction dup a fourni le descripteur **fdDup=4**, première entrée libre dans la table des descripteurs du processus.
- Les descripteurs fd et fdDup pointent sur la même entrée dans la table des fichiers ouverts. En effet, que l'on agisse sur fd ou fdDup et quel que soit le moment où l'on observe les fichiers ouverts par le processus (via lsof), les offsets sont les mêmes pour les deux descripteurs.

Le schéma suivant illustre la situation après l'écriture dans le fichier :



Exemple 3.23 (dup2). Soit le programme suivant

```
***** Test_dup2.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    int fd;
    int fdDup;

    printf("pid : %d\n",getpid());

    if ((fd = open("Sortie.txt",O_RDWR | O_CREAT,0744)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    printf("fd = %d\n",fd);

    if ((fdDup = dup2(fd,1)) == -1)
    {
        perror("Erreur de dup2()");
        exit(1);
    }

    printf("fdDup = %d\n",fdDup);

    sleep(10);
    exit(0);
}
```

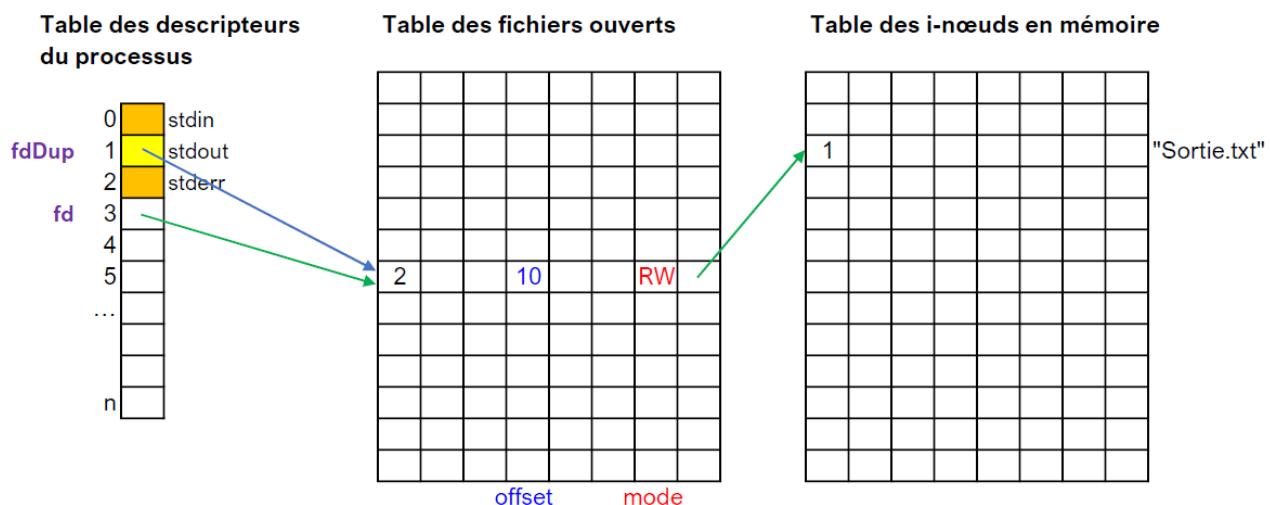
dont un exemple d'exécution fournit

```
# ./Test_dup2 &
[1] 9745
pid : 9745
fd = 3
# lsof -p 9745 -aod "0-10"
COMMAND PID USER FD TYPE DEVICE OFFSET NODE NAME
Test_dup2 9745 student 0u CHR 136,1 0t0 4 /dev/pts/1
Test_dup2 9745 student 1u REG 252,0 0t10 205281960 /home/student/Sortie.txt
Test_dup2 9745 student 2u CHR 136,1 0t0 4 /dev/pts/1
Test_dup2 9745 student 3u REG 252,0 0t10 205281960 /home/student/Sortie.txt
# cat Sortie.txt
fdDup = 1
[1]+ Fini ./Test_dup2
#
```

Nous observons bien que

- L'ouverture du fichier a fourni le descripteur **fd=3**.
- La duplication du descripteur fd par la fonction dup2 a fourni le descripteur **fdDup=1**, descripteur correspondant au STDOUT.
- Le résultat du 3ème printf n'a pas eu lieu dans le terminal mais bien dans le fichier. La sortie standard STDOUT a donc été redirigée vers le fichier Sortie.txt.

Le schéma suivant illustre la situation lors du blocage du processus sur le sleep(10) :



3.3.10 Les primitives “stat” et “fstat”

Cette fonction permet d'obtenir toutes les **informations d'un fichier**. Plus précisément, elle permet de récupérer, dans une variable locale du processus, l'**i-noeud de ce fichier**. Un i-noeud de fichier est représenté par la structure **struct stat** décrite ci-dessous. La documentation peut être obtenue dans le man :

```
# man -s 2 stat
...
#
```

Les prototypes de ces fonctions sont :

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

int stat(const char* path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Cette fonction existe en deux versions mais celles-ci font exactement la même chose. La première reçoit le nom du fichier tandis que la seconde reçoit le descripteur du fichier si celui-ci a été ouvert précédemment.

Paramètres

Les paramètres de ces fonctions sont

Paramètre	Signification
fd	descripteur du fichier déjà ouvert
path	nom du fichier
buf	l'adresse de la structure qui va recevoir les informations sur le fichier

La structure **stat** est quant à elle définie dans le fichier <sys/stat.h> par

```
struct stat {
    dev_t      st_dev;          /* Périphérique */
    ino_t      st_ino;          /* Numéro inœud */
    mode_t     st_mode;         /* Protection */
    nlink_t    st_nlink;        /* Nb liens matériels */
    uid_t      st_uid;          /* UID propriétaire */
    gid_t      st_gid;          /* GID propriétaire */
    dev_t      st_rdev;         /* Type périphérique */
    off_t      st_size;         /* Taille totale en octets */
    blksize_t  st_blksize;      /* Taille de bloc pour E/S */
    blkcnt_t   st_blocks;       /* Nombre de blocs de 512B alloués */
    time_t     st_atime;        /* Heure dernier accès */
    time_t     st_mtime;        /* Heure dernière modification */
    time_t     st_ctime;        /* Heure dernier changement état */
};
```

Voici la signification plus précise de quelques-uns de ces champs :

Champ	Signification
st_mode	contient les informations concernant le type du fichier mais également les droits en lecture/écriture/exécution
st_size	taille totale du fichier en octets
st_atime	heure de dernier accès au fichier (lecture)
st_mtime	heure de dernière modification des données du fichier
st_ctime	heure de dernière modification de l'i-noeud du fichier

Valeurs de retour

Le retour des fonctions **stat** ou **fstat** est

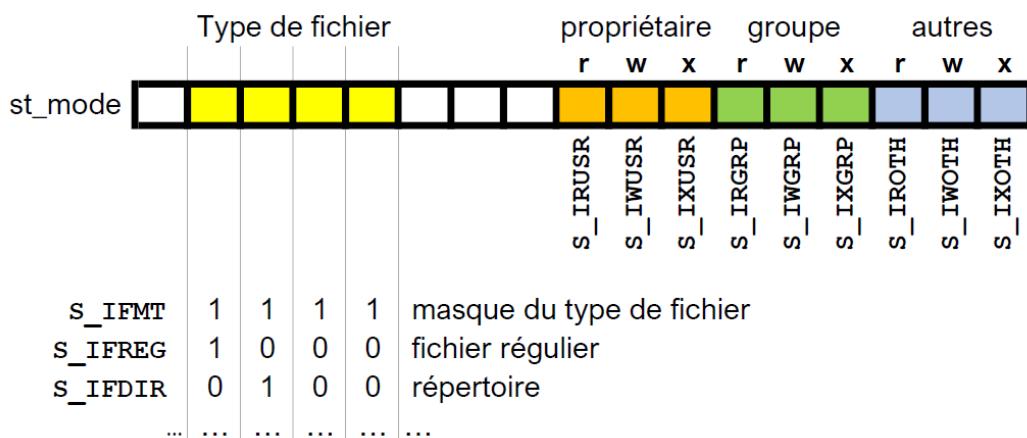
- **0** si la structure a pu être récupérée.
- **-1** en cas d'erreur et, **errno** est positionné.

En cas d'erreur, les valeurs les plus fréquentes de **errno** sont

errno	signification
EACCES	L'accès au fichier n'est pas autorisé.
EFAULT	L'adresse buf n'est pas valide.
EBADF	Le descripteur n'est pas valide.
...	...

Analyse du champ **st_mode**

Le champ **st_mode** de la structure stat définit à la fois le **type de fichier sur 4 bits** et les **droits d'accès au fichier sur 9 bits**. Il faut donc voir le champ **st_mode** comme un "paquet de bits" hors duquel on peut retrouver ces informations à l'aide de masques binaires :



Pour extraire les droits, il suffit d'opérer un "et" logique bit à bit entre le champ **st_mode** et l'un de ces masques (S_IRUSR, ...). L'extraction du type de fichier est un peu plus complexe, et il existe dès lors des **macro-instructions**, prenant en paramètre le champ **st_mode** et permettant facilement de récupérer l'information souhaitée :

Macro-instruction	signification
S_ISREG(m)	retourne vrai si le fichier est un fichier régulier
S_ISDIR(m)	retourne vrai si le fichier est un répertoire
...	...

Exemple 3.24 (stat). Soit le programme suivant

```
***** Test_stat.cpp *****
#include <stdio.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>

int main(int argc,char* argv[])
{
    struct stat StructStat;

    stat(argv[1],&StructStat);

    printf("User: %d\n",StructStat.st_uid);
    printf("Groupe: %d\n",StructStat.st_gid);
    printf("Droits d'accès: %o\n",StructStat.st_mode & 0777);
    if (StructStat.st_mode & S_IWOTH) printf("Accès autorisé en écriture pour les autres\n");
    else printf("Accès non autorisé en écriture pour les autres\n");
    printf("Type: ");
    if (S_ISREG(StructStat.st_mode)) printf("fichier régulier\n");
    if (S_ISDIR(StructStat.st_mode)) printf("fichier répertoire\n");
    printf("Taille: %ld\n",StructStat.st_size);
    printf("Date dernier accès: %s",ctime(&StructStat.st_atime));
    printf("Date dernière modification contenu: %s",ctime(&StructStat.st_mtime));
    printf("Date dernière modification inoeud: %s",ctime(&StructStat.st_ctime));
    printf("No inoeud: %ld\n",StructStat.st_ino);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# id
uid=1000(student) gid=1000(student) ...
# ls -al Essai.txt
-rw-rw-r--. 1 student student 34 12 fév 16:08 Essai.txt
# ./Test_stat Essai.txt
User: 1000
Groupe: 1000
Droits d'accès: 0664
Accès non autorisé en écriture pour les autres
Type: fichier régulier
Taille: 34
Date dernier accès: Fri Feb 12 16:08:48 2021
Date dernière modification contenu: Fri Feb 12 16:08:36 2021
Date dernière modification inoeud: Fri Feb 12 16:08:36 2021
No inoeud: 205281955
# ./Test_stat .
User: 1000
Groupe: 1000
Droits d'accès: 0775
Accès non autorisé en écriture pour les autres
Type: fichier répertoire
Taille: 4096
Date dernier accès: Mon Feb 15 10:43:47 2021
Date dernière modification contenu: Mon Feb 15 10:34:52 2021
Date dernière modification inoeud: Mon Feb 15 10:34:52 2021
No inoeud: 205197316
#
```

3.4 Appels bufferisés ou non bufferisés ?

On pourrait à présent se demander dans quels cas il est préférable d'utiliser l'une ou l'autre techniques d'accès aux fichiers : bufferisée ou non bufferisée ? Ou encore les **flux de données** ou les **descripteurs** ?

Les flux de données

Comme on l'a déjà abordé précédemment, les flux de données sont représentés par la **structure FILE**, et un fichier est manipulé par un pointeur de type FILE*. Cette structure est dite **opaque** car son contenu est masqué au programmeur applicatif. En interne, un flux est associé à un descripteur et dispose en plus d'une mémoire tampon, ainsi que des membres permettant de mémoriser l'état du fichier.

Les descripteurs de fichiers

Il faut comprendre que les descripteurs de fichiers appartiennent à l'**interface du noyau Linux**. Les fonctions open(), close(), read(), write() par exemple sont des appels systèmes qui dialoguent directement avec le noyau Linux (les tables systèmes).

Un modèle en couche

A l'inverse, les flux sont une **couche supérieure** ajoutée aux descripteurs et qui n'appartient qu'à la **bibliothèque C**. Les fonctions fopen(), fclose(), fread() ou fwrite() ne sont implémentées que dans la bibliothèque C et travaillent dans l'espace adressable du processus. Le noyau Linux n'a aucune connaissance de la notion de flux.

Un souci de portabilité

Ceci explique que la **bibliothèque d'entrée/sortie du C standard** ne comporte aucune indication concernant les descripteurs. Ceux-ci sont à l'origine spécifiques aux systèmes d'exploitation de type Unix et ne peuvent pas être pris en considération dans une normalisation générale. Pour assurer la portabilité d'un programme, on préférera donc l'**utilisation des flux** même si la plupart des systèmes d'exploitation courants implémentent les fonctions d'accès aux descripteurs de fichiers.

Programmation système

Comme on l'a vu en début de chapitre, l'utilisation des flux, lorsque l'on travaille en multi-processus peut engendrer des soucis de synchronisation. Dans ce contexte, il sera alors préférable d'utiliser l'accès non bufferisé aux fichiers et les **descripteurs**.

3.4.1 Correspondance entre les fonctions de haut et de bas niveau

Il existe une correspondance directe entre les modes d'ouverture bufferisée et non bufferisée :

Ouverture bufferisée	Ouverture non bufferisée
“r”	O_RDONLY
“w”	O_WRONLY O_CREAT O_TRUNC
“a”	O_WRONLY O_CREAT O_APPEND
“r+”	O_RDWR
“w+”	O_RDWR O_CREAT O_TRUNC
“a+”	O_RDWR O_CREAT O_APPEND

Quelques remarques :

1. Quel que soit le mode choisi, il n'y a pas de distinction entre le mode texte et le mode binaire.
 2. En mode bufferisé, les droits d'un fichier obtenus lors de sa création sont 0666 qui seront ensuite modifiés par la valeur d'umask du processus.

Exemple 3.25 (fopen). Soit le programme suivant

```
***** Test_fopen.cpp *****

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

int main()
{
    FILE* fb;
    if ((fb = fopen("Test.dat","w")) == NULL)
    {
        perror("Erreur de fopen");
        exit(1);
    }

    fclose(fb);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_fopen
# ls -al Test.dat
-rw-r--r--. 1 student student 0 15 fév 16:29 Test.dat
#
```

La valeur de l'umask étant de 022, les droits obtenus pour le fichier sont

```

droits fopen    = 0666   = 110 110 110
& ~ umask      = 0755   = 111 101 101
droits obtenus = 0644   = 110 100 100

```

3.4.2 La fonction “fileo”

Cette fonction permet d'obtenir le **descripteur** d'un fichier qui a été ouvert précédemment en **mode bufferisé** à l'aide de la fonction fopen. La documentation peut être obtenue dans le

man :

```
# man fileno
...
#
```

Le prototype de cette fonction est :

```
#include <stdio.h>

int fileno(FILE* stream);
```

Paramètre

Le paramètre de cette fonction est

Paramètre	Signification
stream	le flux FILE* du fichier ouvert en bufferisé

Valeur de retour

Le retour de la fonction **fileno** est **une valeur entière positive** correspondant au descripteur du fichier.

En cas d'erreur (stream ne correspondant pas à un fichier ouvert correctement), une erreur fatale se produit (erreur de segmentation).

Exemple 3.26 (fileno). Soit le programme suivant

```
***** Test_fileno.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    FILE* fb;
    int fd;

    printf("pid : %d\n",getpid());
    if ((fb = fopen(argv[1],"w")) == NULL)
    {
        perror("Erreur de open()");
        exit(1);
    }

    fd = fileno(fb);

    printf("Descripteur associe a %s : %d\n",argv[1],fd);

    sleep(10);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_fileno aaa.dat &
[1] 15752
pid : 15752
Decripteur associe a aaa.dat : 3
# lsof -p 15752 -ad "0-10"
COMMAND      PID    USER FD   TYPE DEVICE SIZE/OFF          NODE NAME
Test_file 15752 student  0u    CHR  136,1      0t0            4 /dev/pts/1
Test_file 15752 student  1u    CHR  136,1      0t0            4 /dev/pts/1
Test_file 15752 student  2u    CHR  136,1      0t0            4 /dev/pts/1
Test_file 15752 student  3w    REG   252,0      0 205281969 /home/student/aaa.dat
#
```



Exemple 3.27 (Redirection du stderr vers un fichier de trace). Soit le programme suivant

```
***** Redir_stderr.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    FILE* fb;
    int fd, fdDup;

    printf("pid : %d\n",getpid());

    if ((fb = fopen(argv[1],"w")) == NULL)
    {
        perror("Erreur de fopen()");
        exit(1);
    }

    fd = fileno(fb);

    printf("Decripteur associe a %s : %d\n",argv[1],fd);

    if ((fdDup = dup2(fd,2)) == -1)
    {
        perror("Erreur de dup2()");
        exit(1);
    }

    printf("Decripteur duplique : %d\n",fdDup);

    printf("Sur le stdout\n");
    perror("Sur le stderr\n");

    sleep(10);

    printf("\nfin du programme\n");
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Redir_stderr Trace.log &
[1] 17375
pid : 17375
Descripteur associe a Trace.log : 3
Descripteur duplique : 2
Sur le stdout
# lsof -p 17375 -ad "0-10"
COMMAND      PID    USER FD   TYPE DEVICE SIZE/OFF        NODE NAME
Redir_std 17375 student  0u    CHR  136,1      0t0          4 /dev/pts/1
Redir_std 17375 student  1u    CHR  136,1      0t0          4 /dev/pts/1
Redir_std 17375 student  2w    REG  252,0      24 205281973 /home/student/Trace.log
Redir_std 17375 student  3w    REG  252,0      24 205281973 /home/student/Trace.log
#
fin du programme
[1]+ Fini ./Redir_stderr Trace.log
# cat Trace.log
Sur le stderr
: Success
#
```



3.4.3 La fonction “fdopen”

Cette fonction permet d'obtenir un **flux de données** (FILE*) à partir du **descripteur** d'un fichier déjà ouvert en non bufferisé. L'accès au fichier pourra alors se faire par le descripteur ou le flux qui sont tous deux associés à la **même entrée dans la table des fichiers ouverts**. La documentation peut être obtenue dans le man :

```
# man fdopen
...
#
```

Le prototype de cette fonction est :

```
#include <stdio.h>

FILE* fdopen(int fd, const char* mode);
```

Paramètres

Les paramètres de cette fonction sont

Paramètre	Signification
fd	le descripteur du fichier déjà ouvert en non bufferisé
mode	le mode d'ouverture

Le mode d'ouverture (“r”, “r+”, ...) doit être compatible avec le mode d'ouverture non bufferisée.

Valeurs de retour

Le retour de la fonction **fdopen** est

- **un pointeur de structure FILE*** associé au descripteur fd.
- **NULL** en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EINVAL	Le mode d'ouverture n'est pas valide.
...	...

Exemple 3.28 (fdopen). Soit le programme suivant

```
***** Test_fdopen.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd;
    FILE* fb;

    printf("pid : %d\n",getpid());
    if ((fd = open(argv[1],O_CREAT | O_WRONLY,0644)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    if ((fb = fdopen(fd,"w")) == NULL)
    {
        perror("Erreur de fdopen()");
        exit(1);
    }

    fwrite("EnPremier\n",1,10,fb);
    write(fd,"EnSecond\n",9);

    sleep(10);

    fclose(fb);

    if (close(fd))
    {
        perror("\nErreur de close()");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_fdopen BBB.txt &
[1] 22825
pid : 22825
# lsof -p 22825 -ad "0-10"
COMMAND      PID    USER   FD   TYPE DEVICE SIZE/OFF          NODE NAME
Test_fdop 22825 student  0u   CHR  136,1      0t0            4 /dev/pts/1
Test_fdop 22825 student  1u   CHR  136,1      0t0            4 /dev/pts/1
Test_fdop 22825 student  2u   CHR  136,1      0t0            4 /dev/pts/1
Test_fdop 22825 student  3w   REG  252,0      9 205281984 /home/student/BBB.txt
#
Erreur de close(): Bad file descriptor
[1]+  Termine 1      ./Test_fdopen BBB.txt
# cat BBB.txt
EnSecond
EnPremier
#
```

Plusieurs commentaires :

1. Au moment de l'exécution du lsof, le processus est bloqué sur le sleep(10). On remarque que la taille du fichier est alors de 9 bytes. Cela signifie que les 9 bytes du write() ont été écrits sur disque tandis que les 10 bytes du fwrite() sont toujours dans le buffer mémoire pointé par fb.
2. L'affichage du contenu du fichier confirme la première remarque. Le buffer fb a été vidé lors du fclose(). Dès lors, "EnPremier" a été écrit après "EnSecond".
3. L'appel de close() a provoqué une erreur. En effet, le fclose() a déjà fermé le fichier et le descripteur associé a déjà été libéré.



Cet exemple nous montre à nouveau qu'il est préférable de ne pas manipuler un fichier simultanément en bufferisé et non bufferisé.

Chapitre 4

Les processus

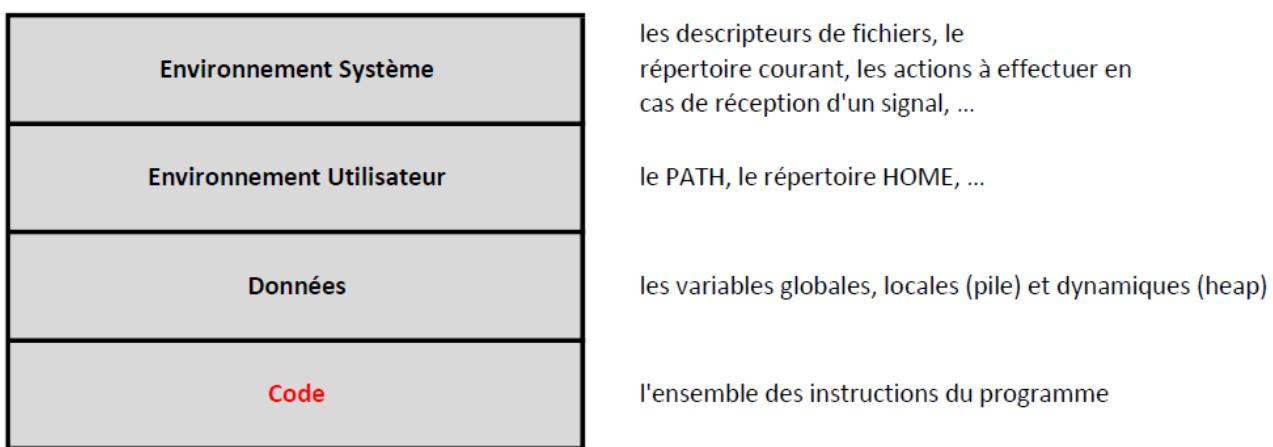
4.1 Introduction

4.1.1 Définition, structure et identifiant d'un processus

Il est important de bien distinguer les notions de **programme** et de **processus** :

- Un **programme** est un **fichier exécutable** : c'est par exemple le résultat d'une compilation.
- Un **processus** est l'**image mémoire d'un programme en cours d'exécution**.

Lorsqu'on lance un programme en ligne de commande, une zone mémoire lui est réservée et on peut dire que cet espace mémoire porte le nom de processus. Schématiquement, cet espace mémoire peut être structuré selon



Les données contenues dans les zones "Environnement Système" et "Environnement Utilisateur" sont implicites (non déclarées et définies par le programmeur) tandis que les zones "Données" et "Code" correspondent au code source du développeur.

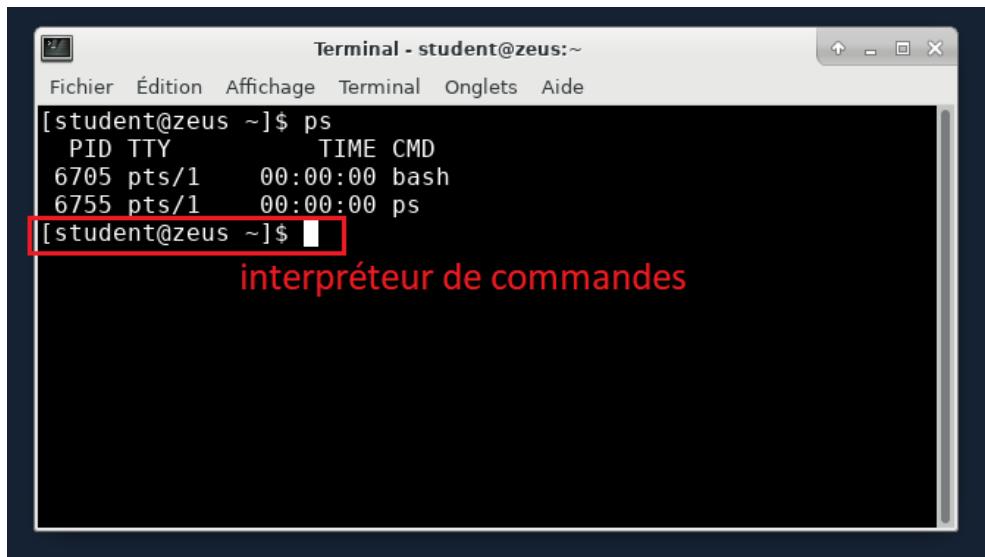
Une fois créé, un processus est identifié sur le système par un entier positif appelé **PID** ("Process IDentifier"). Cet entier **unique** lui est **attribué par le système** au moment de sa création, et il le conserve tout au long de sa vie.

A l'exception du tout premier processus lancé sur un système Linux (appelé processus **init** et de **PID** égal à **1**), un processus est forcément créé par un autre processus qui est qualifié de

père du processus. L'identifiant du père d'un processus est noté **PPID** pour “**Parent Process Identifier**”.

4.1.2 L'interpréteur de commandes

Un processus important est l'**interpréteur de commandes** (on parle également plus simplement du “**shell**”) dont le rôle d'interpréter et d'exécuter une commande tapée par l'utilisateur. Il est lancé lorsqu'un terminal est ouvert. Par exemple :



Pour s'en rendre compte, il suffit de taper la commande **ps** qui affiche tous les processus lancés dans le terminal. Dans cet exemple, on remarque que

1. l'exécution de la **commande ps** a donné naissance à un **processus** appelé **ps** dont le **PID** est égal à 6755. Il s'agit de l'image mémoire du programme **ps** en cours d'exécution dans ce terminal.
2. l'**interpréteur de commandes** est un processus appelé **bash** dont le **PID** est égal à 6705.

Dans cet exemple, l'interpréteur de commandes est un processus issus de l'**exécutable /bin/bash**. Comme déjà vu au chapitre précédent, celui-ci est spécifié par utilisateur dans le fichier **/etc/passwd**. Mais d'autres interpréteurs de commandes existent, avec leurs spécificités propres :

- /bin/sh
- /bin/bash
- /bin/csh
- ...

L'interpréteur de commandes étant lui même un processus, il possède une zone mémoire “Environnement Utilisateur” contenant notamment les variables d'environnement **PATH** et **HOME**. C'est grâce à la variable **PATH** que l'interpréteur de commandes est capable de trouver l'emplacement du fichier exécutable qu'il doit exécuter :

```
# which ps
/usr/bin/ps
# echo $PATH
/usr/local/jdk1.8.0_111/bin:/opt/rh/devtoolset-6/root/usr/bin:/usr/lib64/qt-3.3/bin:
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:
/home/student/.local/bin:/home/student/bin
# echo $HOME
/home/student
#
```

Plusieurs remarques :

1. la commande `which` permet de savoir où se situe le fichier `ps` qui est exécuté.
2. le chemin d'accès à la commande `ps` (`/usr/bin`) se trouve dans la variable `PATH` de l'interpréteur de commandes. Voilà pourquoi il n'est pas nécessaire de taper le chemin d'accès complet à la commande. Le `shell` parcourt en effet tous les chemins d'accès disponibles dans sa variable `PATH` à la recherche de la commande `ps`. Dès qu'elle est trouvée, elle est exécutée.
3. Dans cet exemple, la variable `PATH` ne contient pas le **répertoire courant** . parmi ses chemins d'accès. Il sera donc nécessaire de faire précéder le nom d'un exécutable du répertoire courant par `./` pour pouvoir l'exécuter.
4. L'utilisateur courant est `student` car le répertoire `HOME` est `/home/student`.

4.1.3 Lancement synchrone et asynchrone d'un processus

Le shell peut exécuter un processus, par interprétation d'une commande, de deux manières :

1. De manière **synchrone** : le `shell` attend que l'exécution de la commande soit terminée avant de rendre la main à l'utilisateur. Il y a donc ce que l'on appelle "**synchronisation**" entre le processus shell et le processus créé lors de l'exécution de la commande.
2. De manière **asynchrone** : dès que le processus est lancé, le shell rend directement la main à l'utilisateur. Les deux processus, shell et celui qui exécute la commande, s'exécutent alors en "**parallèle**", c'est-à-dire "**en même temps**". Pour lancer une commande de manière asynchrone, il faut placer la caractère `&` à la fin de la commande. On dit également d'un processus lancé de manière asynchrone qu'il est lancé **en arrière plan**.

Exemple 4.1 (lancement synchrone et asynchrone d'un processus). Soit le programme suivant :

```
***** Prog.cpp *****
#include <stdlib.h>
#include <unistd.h>

int main()
{
    sleep(5);
    exit(1);
}
```

dont quelques exemples d'exécution fournissent

```
# ./Prog
# ps
 PID TTY      TIME CMD
 6705 pts/1  00:00:00 bash
 8202 pts/1  00:00:00 ps
# ./Prog &
[1] 8210
# ./Prog &
[2] 8217
# ps
 PID TTY      TIME CMD
 6705 pts/1  00:00:00 bash
 8210 pts/1  00:00:00 Prog
 8217 pts/1  00:00:00 Prog
 8232 pts/1  00:00:00 ps
# ps
 PID TTY      TIME CMD
 6705 pts/1  00:00:00 bash
 8239 pts/1  00:00:00 ps
[1]- Termine 1      ./Prog
[2]+ Termine 1      ./Prog
#
```

Plusieurs commentaires :

1. Le premier processus Prog est lancé de manière **synchrone** à partir du fichier exécutable ./Prog situé dans le répertoire courant. Un laps de temps de 5 secondes se passe avant de pouvoir lancer la commande ps pour la première fois. En effet, le **shell attend** que Prog se termine avant de rendre la main à l'utilisateur.
2. Deux processus Prog sont créés à partir du même programme ./Prog et sont lancés de manière **asynchrone**. La commande ps permet de voir que les deux processus créés ont des PID valant 8210 et 8217.
3. Lors de la dernière exécution de la commande ps, les deux processus Prog sont terminés.



4.1.4 Table des processus et cycle de vie d'un processus

Comme pour les fichiers ouverts, une **table système** gère l'ensemble des processus de la machine. Cette table porte le nom de **table des processus**. Pour chaque entrée de cette table (et donc pour chaque processus), les informations essentielles sont

1. l'**état** du processus,
2. les **adresses** vers les différentes zones mémoires du processus,
3. l'**UID** du processus, c'est-à-dire l'utilisateur propriétaire du processus (par défaut, il s'agit de l'utilisateur qui a créé le processus en exécutant une commande),
4. le **PID** et le **PPID** du processus,
5. l'ensemble des **signaux d'interruptions reçus** mais non encore traités par le processus (voir chapitre sur les signaux),
6. les **priorités**, c'est-à-dire les paramètres pris en compte par le "**scheduler**" pour lui attribuer du temps machine,

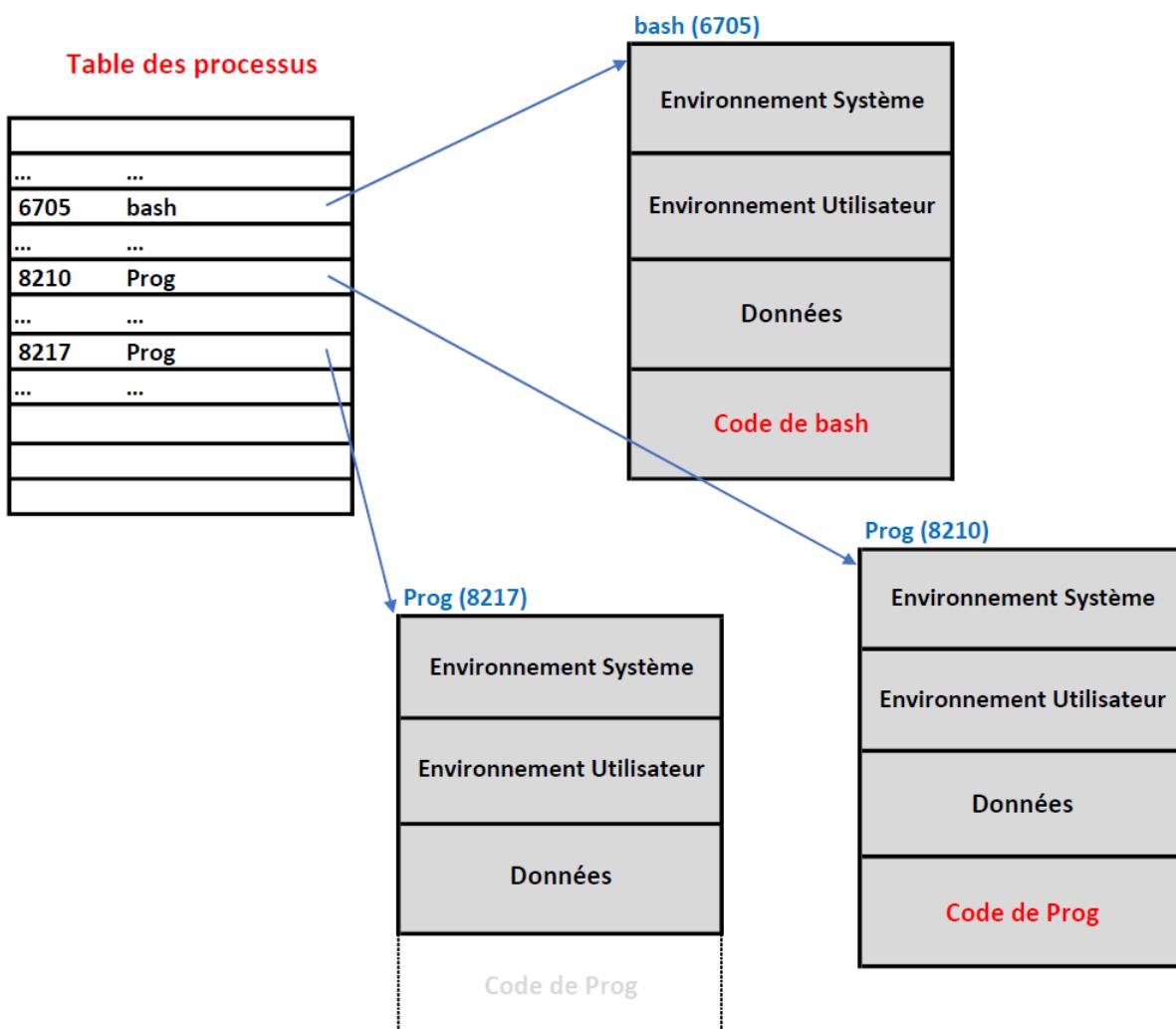
7. les **informations de terminaison** du processus (manière dont il s'est terminé, valeur de retour, ...).

On distingue deux types de processus :

1. les **processus système** qui ont comme propriétaire (UID) le superutilisateur **root**. Ils sont également appelés **processus démons** (déformation de l'anglais "daemon" qui provient lui-même de "Disk And Extension MONitor") et tournent indéfiniment dans le système pour répondre aux requêtes des utilisateurs ou d'autres processus démons. Le processus **init** fait partie des processus démons. Parmi les processus démons, on peut citer également les processus inetd, nfsd, telnetd, lpd, ...
2. les **processus utilisateur** créés à partir des fichiers exécutables des utilisateurs.

L'**ordonnanceur** ou le "**scheduler**" est le mécanisme du système d'exploitation Linux qui permet d'attribuer du "temps machine (ou temps CPU)" au différents processus de la machine. C'est lui qui décide d'attribuer un processeur à un processus pour un laps de temps donné et de lui retirer afin de l'attribuer à un autre processus.

Exemple 4.2 (Table des processus). Reprenons l'**exemple 4.1** où 2 processus Prog ont été lancés de manière asynchrone. Schématiquement, voici l'état de la mémoire à un moment où les deux processus Prog sont toujours en exécution :



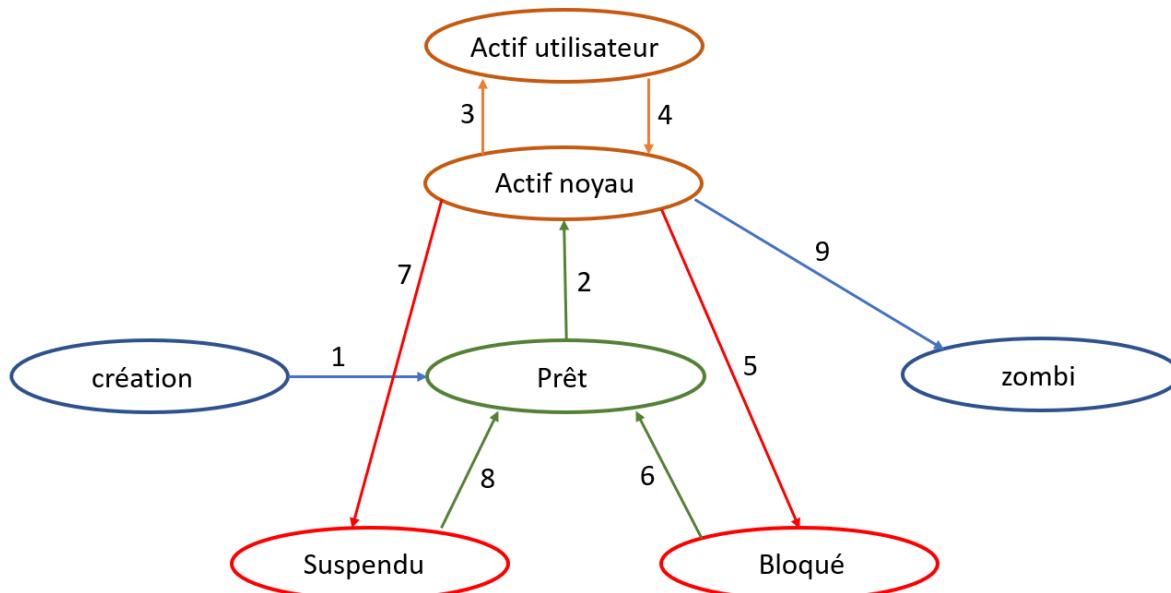
Les deux processus Prog, issus de l'exécutable ./Prog :

1. occupent chacun une entrée distincte dans la table des processus mais sont identifiés de manière unique grâce à leur PID respectifs 8210 et 8217.
2. **partagent le même code**, ce qui signifie que le code de ./Prog n'est présent qu'**une seule fois en mémoire** mais est utilisé par les deux processus en même temps, et cela sans poser de problème, car agissant sur les données propres de chaque processus.

■

Sous Linux, on dit que les processus sont **réentrants**, ou exécutent du **code réentrant**, c'est-à-dire du code qui est présent une seule fois en mémoire et qui peut être exécuté par plusieurs processus simultanément sans poser de problème.

Les processus utilisateur ont une durée de vie limitée : ils sont créés par un processus, exécutent leur code et se terminent. Mais tout au long de leur vie, ils passent par différents **états** repris dans le **cycle de vie** suivant :



A chaque "bulle" correspond un **état d'un processus** et les différentes flèches représentent les transitions possibles entre les états :

1. le processus vient d'être créé par son processus père. Il est **prêt** à s'exécuter mais cela ne veut pas dire qu'il va s'exécuter de suite car il se peut qu'il ne dispose pas d'un processeur pour cela.
2. le processus est "élu" par le **scheduler** qui lui attribue un processeur. Le processus s'exécute alors soit en **mode noyau** (il exécute le code correspondant à un appel système comme read, open, ...), soit en **mode utilisateur** (il exécute des instructions ou des fonctions codées par l'utilisateur).
3. Le processus revient d'un appel système, c'est-à-dire qu'il sort du code d'un appel système (ou d'une interruption, voir chapitre sur les signaux) pour revenir aux instructions de l'utilisateur.
4. le processus a fait un **appel système** ou a été interrompu.
5. le processus ne dispose pas de toutes les ressources pour pouvoir continuer son exécution et se met **en attente** d'un événement ou des ressources nécessaires.

6. l'événement attendu s'est produit ou les ressources nécessaires sont à présent disponibles.
7. le processus est **suspendu** par la réception du signal SIGSTOP (voir chapitre sur les signaux).
8. le processus est **réveillé** par un signal SIGCONT et peut poursuivre son exécution.
9. le processus arrive à la fin de son code ou est interrompu, et son exécution **se termine**.

Lorsqu'un processus atteint la fin de son code ou est interrompu, il se termine en passant à l'état **zombi**, ce qui signifie que

1. la zone mémoire réservée pour lui (environnement système et utilisateur, données et code) est libérée, mais
2. son entrée **reste présente dans la table des processus**. Celle-ci contient alors toutes les informations sur la **fin du processus**. Cette entrée restera active tant que le père du processus ne l'aura supprimée de la table des processus. S'il s'agit d'un processus lancé en ligne de commande, c'est le processus **shell** qui va le supprimer de la table des processus.

4.1.5 Identité et propriétaires d'un processus

Comme nous l'avons déjà vu, un processus Linux est identifié sur le système par un entier positif unique appelé **PID** (Process Identifier). Cet identifiant lui est attribué lors de sa création et il le conserve tout au long de sa vie. Comme nous l'avons vu également, tout processus utilisateur est créé par son processus **père** qui possède également son propre PID.

Il est possible pour un processus de connaître le **PID** qui lui a été **attribué**, ainsi que le **PID de son père**. Pour cela, il peut utiliser les appels systèmes suivants :

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

pid_t getpid(void);
pid_t getppid(void);
```

Le type **pid_t**, représentant le **PID** d'un processus, est défini dans le fichier sys/types.h et correspond à un **entier signé**. La fonction

- **getpid()** retourne le PID du processus appelant,
- **getppid()** le retourne le PID du père du processus appelant.

Ces fonctions n'échouent jamais.

Exemple 4.3 (getpid et getppid). Soit le programme suivant :

```
***** Test_getpid.cpp *****/
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Mon PID = %d\n",getpid());
```

```

printf("PID de mon pere = %d\n",getppid());

sleep(5);
exit(1);
}

```

dont un exemple d'exécution fournit

```

# ./Test_getpid &
[1] 18496
Mon PID = 18496
PID de mon pere = 6705
# ps
  PID TTY          TIME CMD
  6705 pts/1    00:00:00 bash
18496 pts/1    00:00:00 Test_getpid
18505 pts/1    00:00:00 ps
#

```

Plusieurs commentaires :

1. Le processus **Test_getpid** étant lancé de manière **asynchrone**, la commande **ps** a pu être lancée pendant son exécution.
2. Le père du processus **Test_getpid** ayant un PID égal à 6705, il s'agit du processus **bash**. C'est donc bien le processus shell qui a créé le processus **Test_getpid**.



Un système Linux est particulièrement orienté vers l'identification de ses utilisateurs. Toute activité entreprise par un utilisateur est soumise à des contrôles par rapport aux permissions qui lui sont attribuées. Pour cela,

- tout fichier sur disque est la propriété d'un utilisateur, mais
- tout processus est également propriété d'un utilisateur, ce qui lui confère les droits liés à cet utilisateur. Il s'agit de l'**UID** ("User IDentifier") du processus.

On distingue deux propriétaires différents pour un processus :

1. le **propriétaire réel** : il s'agit de l'utilisateur qui **a créé** le processus (par lancement dans un shell).
2. le **propriétaire effectif** : il s'agit de l'utilisateur propriétaire du processus **à un moment donné** de son exécution.

Au démarrage d'un processus, les propriétaires réel et effectif sont identiques mais un processus peut **changer** de **propriétaire effectif** à un moment donné afin, par exemple, d'obtenir certains droits que l'utilisateur de départ ne possède pas.

Il est possible pour un processus de connaître son utilisateur propriétaire. Pour cela, il peut utiliser les appels systèmes suivants :

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

uid_t getuid(void);
uid_t geteuid(void);

gid_t getgid(void);
gid_t getegid(void);
```

Les types **uid_t** et **gid_t**, représentant respectivement les identifiants d'un utilisateur et d'un groupe d'utilisateurs, sont définis dans le fichier sys/types.h et correspondent à des **entiers signés**. La fonction

- **getuid()** retourne le **propriétaire réel (UID)** du processus appelant,
- **geteuid()** retourne le **propriétaire effectif (EUID)** du processus appelant,
- **getgid()** retourne le groupe du propriétaire réel du processus appelant.
- **getegid()** retourne le groupe du propriétaire effectif du processus appelant.

Ces fonctions n'échouent jamais.

Exemple 4.4 (getuid, geteuid, getgid et getegid). Soit le programme suivant :

```
***** Test_getuid.cpp *****/
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("UID=%d ",getuid());
    printf("EUID=%d ",geteuid());
    printf("GID=%d ",getgid());
    printf("EGID=%d\n",getegid());

    exit(1);
}
```

dont quelques exemples d'exécution fournissent

```

# id
uid=1000(student) gid=1000(student) groupes=1000(student),10(wheel) ...
# ./Test_getuid
UID=1000 EUID=1000 GID=1000 EGID=1000
# su wagner
Mot de passe :
# id
uid=500(wagner) gid=500(wagner) groupes=500(wagner) ...
# ./Test_getuid
UID=500 EUID=500 GID=500 EGID=500
# exit
exit
# chmod +s Test_getuid
# ls -l Test_getuid
-rwsrwsr-x. 1 student student 8760 12 mai 16:49 Test_getuid
# ./Test_getuid
UID=1000 EUID=1000 GID=1000 EGID=1000
# su wagner
Mot de passe :
# ./Test_getuid
UID=500 EUID=1000 GID=500 EGID=1000
#

```

Plusieurs commentaires :

1. la commande **id** permet de connaître l'**identité de l'utilisateur actuel**, tandis que la commande **su** permet de **changer l'utilisateur courant**. L'utilisation de la commande **exit** a permis de retourner à l'utilisateur initial (student).
2. La première exécution de **./Test_uid** a été réalisée en tant que l'utilisateur **student (UID=1000)**. Il est donc créateur du processus Test_uid correspondant et rien n'a été fait pour que le processus change de propriétaire effectif, donc l'**EUID** est égal à **1000** également.
3. La seconde exécution de **./Test_uid** a été réalisée en tant que l'utilisateur **wagner (UID=500)**. Il est donc créateur du processus Test_uid correspondant et rien n'a été fait pour que le processus change de propriétaire effectif, donc l'**EUID** est égal à **500** également.
4. Le propriétaire du fichier exécutable **./Test_getuid** (student) a modifié les droits du fichier en utilisant la commande **chmod** avec l'option **+s**. Ceci a eu pour effet de modifier le droit **x** pour l'utilisateur et pour le groupe en **s**. Cela signifie que le processus va s'exécuter sous la **propriété effective du propriétaire du fichier exécutable et non de l'utilisateur actuel**. Cela s'appelle le **mécanisme de substitution d'identité**. Ceci ne peut se faire qu'avec l'acceptation du propriétaire du fichier exécutable qui, seul, peut en modifier les droits.
5. La dernière exécution de **./Test_getuid** a été réalisée en tant que l'utilisateur **wagner (UID=500)**. Il est donc créateur du processus Test_getuid. Le mécanisme de substitution d'identité fait que le processus devient immédiatement la propriété effective de **student (EUID=1000)**, propriétaire du fichier **./Test_getuid**.
6. Le même raisonnement peut être fait à propos des **groupes** utilisateurs (**GID** et **EGID**). Le droit **+s** peut être spécifié pour l'utilisateur et le groupe en même temps (comme cela est fait ici) mais également séparément avec l'option **u+s** et **g+s** de la commande **chmod**.



4.2 Crée ses propres processus

4.2.1 Mécanisme de lancement d'une commande par le shell

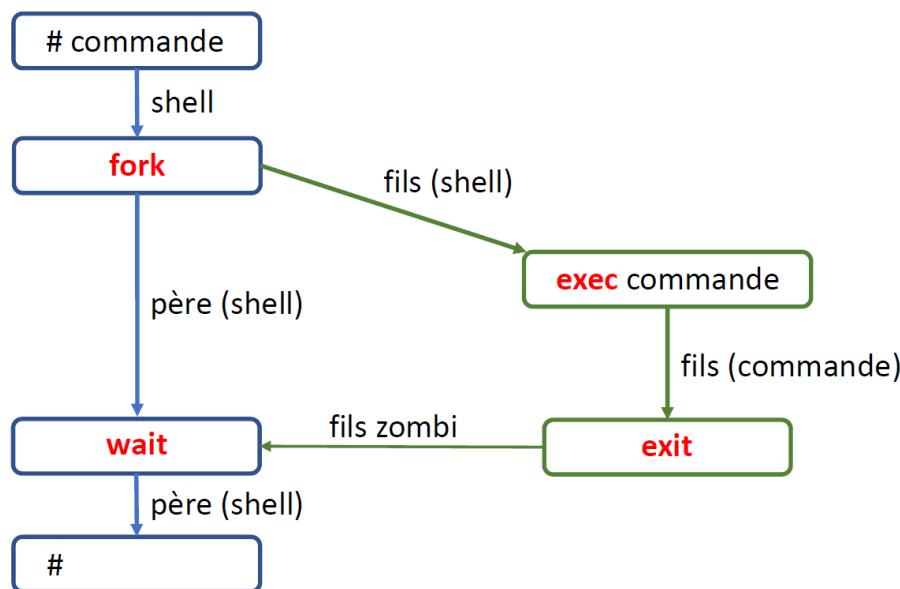
Au moment où une commande est lancée, le processus shell, appelé processus **père**, crée un nouveau processus qualifié de processus **fils**. Pour cela, processus père et processus fils utilisent les 4 appels systèmes suivants :

1. **fork**,
2. **exec**,
3. **exit**, et
4. **wait**.

Ces 4 fonctions seront étudiées en détails dans la suite.

Synchronisation

Dans le cas où la commande est lancée de manière **synchrone**, le schéma général de création est le suivant :



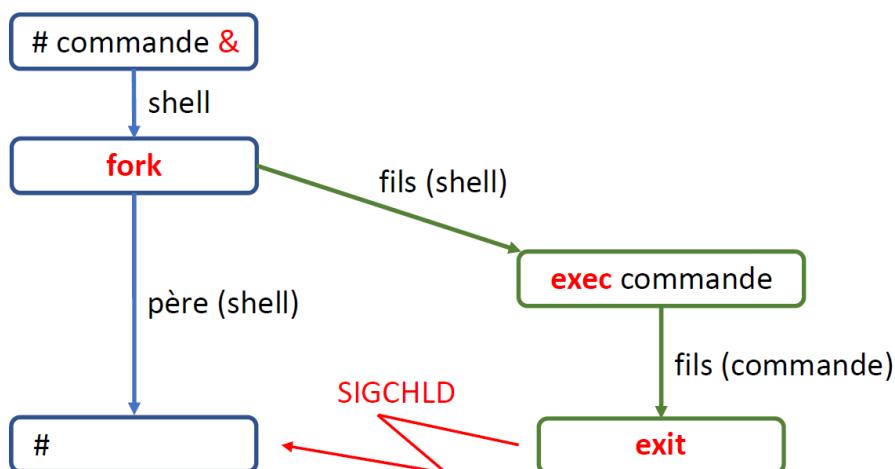
1. Le shell fait appel à la fonction **fork**, ce qui à pour effet de **duplicer le processus appelant**. Cette duplication consiste en une recopie en mémoire de l'environnement système et utilisateur, et des données du processus shell, mais pas du code car celui-ci est commun aux deux processus. Il y a en effet **création d'un nouveau processus** qui est une **copie conforme** du processus shell père.
2. Le processus shell père fait alors appel à la fonction **wait** qui est bloquante et qui attend la fin du processus fils.
3. Le processus shell fils fait appel à la fonction **exec**, ce qui a pour effet de **recouvrir le processus appelant** par l'environnement système et utilisateur, les données et le code de la commande. A ce stade, il n'y a pas de création d'un nouveau processus. En quelque sorte, le processus fils shell "se transforme" en "commande" et exécute le code de la commande.

4. Le processus fils (commande) s'exécute alors jusqu'à arriver à la fin de son code et réaliser l'appel de la fonction **exit**. Ceci a pour effet de **terminer le processus** fils qui devient **zombi**, et de réveiller le processus shell père.
5. Le processus shell père revient de son appel système **wait**, après avoir **supprimé** son processus fils de la **table des processus**.
6. Le processus shell se remet en attente d'une commande de l'utilisateur.

Ici, on dit qu'il y a **synchronisation** entre processus fils et processus père. Le shell est bloqué (on dit aussi endormi) sur l'appel système jusqu'au moment où le processus fils se termine.

Asynchronisme

Dans le cas où la commande est lancée de manière **asynchrone**, le schéma général de création est le suivant :



Le principe général reste le même, mais

1. Le processus shell père continue son exécution après avoir créé le processus fils, il peut donc interpréter d'autres commandes pendant que le processus fils s'exécute **en parallèle**.
2. Lors de son appel à la fonction **exit**, le processus fils va prévenir son processus père qu'il vient de se terminer. Pour cela, il envoie le **signal d'interruption SIGCHLD** à son processus père avant de devenir zombi.
3. De manière **asynchrone** (par la réception du signal SIGCHLD), le processus père est averti que son processus fils vient de se terminer. Il va donc le **supprimer de la table des processus** en faisant appel à la fonction **wait**.
4. Le processus shell se remet ensuite en attente d'une commande utilisateur.

La grosse différence réside ici dans l'**asynchronisme**. Le processus père ne sait absolument pas quand son processus fils va se terminer. Il est donc nécessaire de le prévenir en lui envoyant le signal **SIGCHLD**.

D'une manière générale,

Toute processus qui **se termine** (d'une manière ou d'une autre) **envoie le signal SIGCHLD à son processus père**. Le rôle du processus père est alors de **supprimer son processus fils de la table des processus**, et cela via l'appel à la fonction **wait**.

Même lorsque la commande est lancée de manière **synchrone**, le processus fils envoie le signal **SIGCHLD** au processus shell qui l'a créé. C'est la réception du signal SIGCHLD qui provoque le réveil du processus shell qui était endormi sur l'appel système wait.

4.2.2 Crédit d'un processus : l'appel système fork

La **seule manière** de créer un processus est d'utiliser l'appel système **fork**. Tous les processus Linux sont créés de cette manière et programmer la création de nouveaux processus devra passer par cette méthode.

La fonction fork crée un **nouveau processus** (appelé processus **fils**) qui est la **copie exacte** du processus appelant (appelée processus **père**). Le processus fils ainsi créé est un processus **indépendant** qui possède son **propre PID**. En d'autres termes, la vie du processus créé n'est pas conditionnée par celle de son père, la fin du processus père n'entraîne pas la fin du processus fils. Lors de la création du processus fils, tout est dupliqué à l'**exception du code** qui est **partagé** par les deux processus.

A sa création, un processus fils possède le même environnement utilisateur et système que son père (une copie donc). Dès lors, processus père et fils ont

- les **mêmes propriétaires** UID, EUID, GID et EGID,
- les **mêmes fichiers ouverts**,
- les **mêmes signaux armés** (voir chapitre sur les signaux),
- les **mêmes registres** et donc le même registre d'instruction suivante,
- ...

On dit que le processus fils **hérite des attributs** de son père. La documentation de la fonction fork peut être obtenue dans le man :

```
# man fork
...
#
```

Le prototype de cette fonction est :

```
#include <unistd.h>
pid_t fork(void);
```

Paramètre

Cette fonction n'admet **aucun paramètre**. En effet, la duplication du processus n'en a pas besoin.

Valeurs de retour

Au retour de la fonction fork, nous sommes donc en présence de deux processus et tout se passe comme si les deux processus venaient d'exécuter la fonction fork. Donc, le retour de la fonction **fork** est

- dans le cas du **père** : **une valeur entière positive** correspondant au **PID du fils** qui vient d'être créé.
- dans le cas du **fils** : la valeur entière **0**.
- **-1** en cas d'erreur et, errno est positionné.

En cas d'erreur, les valeurs les plus fréquentes de errno sont

errno	signification
EAGAIN	Le nombre maximum de processus est atteint.
ENOMEM	Pas assez d'espace mémoire.
...	...

Exemple 4.5 (fork). Soit le programme suivant

```
***** Test_fork.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    pid_t id;

    printf("Mon PID = %d\n",getpid());
    printf("Mon PPID = %d\n",getppid());
    printf("Creation d'un processus fils...\n");

    id = fork();
    printf("PID=%d \t PPID=%d \t retour de fork = %d\n",getpid(),getppid(),id);

    sleep(1);

    exit(0);
}
```

dont un premier exemple d'exécution fournit

```
# ps
  PID TTY      TIME CMD
 1091 pts/1    00:00:00 ps
 6705 pts/1    00:00:00 bash
# ./Test_fork
Mon PID = 1099
Mon PPID = 6705
Creation d'un processus fils...
PID=1099      PPID=6705      retour de fork = 1100
PID=1100      PPID=1099      retour de fork = 0
#
```

On constate que :

1. Le père du processus **Test_fork** est bien le **shell** dont le PID est 6705.
2. L'appel de la fonction **fork** par le processus **Test_fork** (de PID égal à 1099) a donné naissance au processus fils dont le PID est 1100.
3. Le **retour** de la fonction fork pour le processus père est **1100 (PID du fils)** et **0** pour le processus fils.
4. Lorsque le processus fils démarre, il **ne recommence pas au début** de la fonction main mais bien **juste après l'appel de la fonction fork**. On remarque également que les deux processus exécutent le même code.

Si on met l'appel de **sleep(1)** en commentaire dans le code, une nouvelle exécution du programme fournit :

```
# echo $$
6705
# ./Test_fork
Mon PID = 1538
Mon PPID = 6705
Creation d'un processus fils...
PID=1538      PPID=6705      retour de fork = 1539
PID=1539      PPID=1          retour de fork = 0
#
```

On observe à présent que

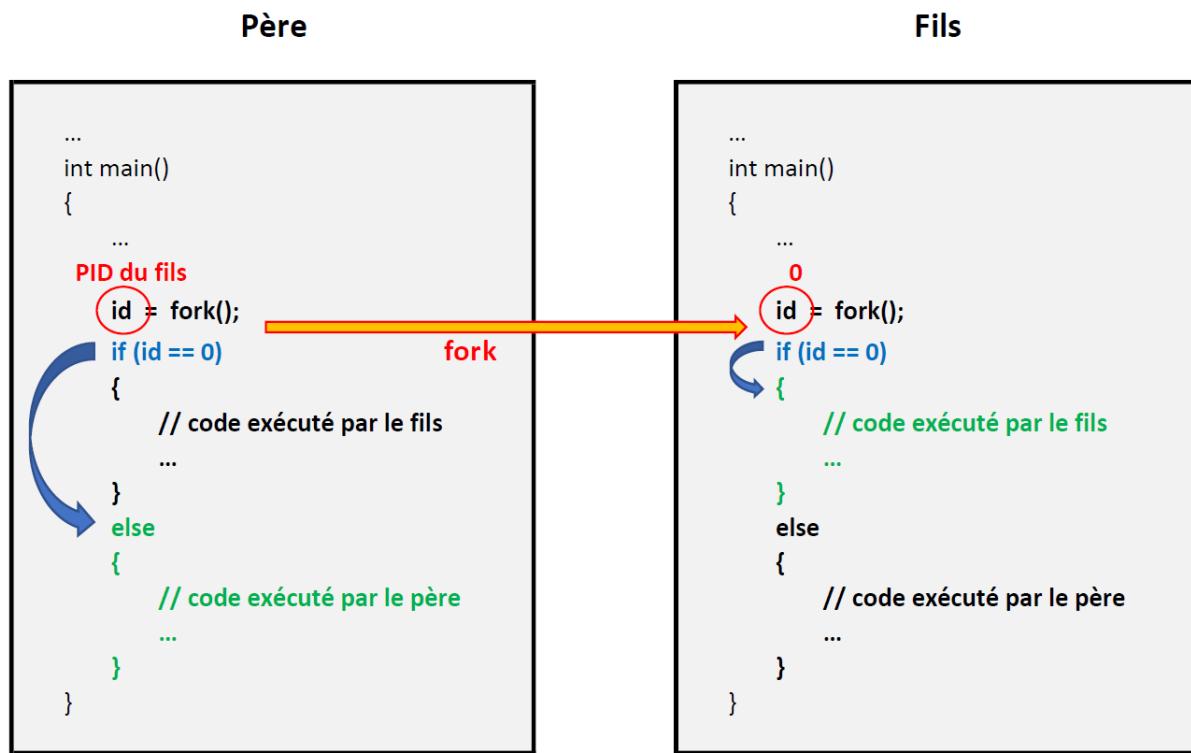
1. l'appel de la commande **echo \$\$** permet d'interroger le **shell** afin qu'il affiche le contenu de sa **variable d'environnement \$** qui contient **son PID** (6705).
2. Le processus fils de PID 1539 créé par le processus **Test_fork** de PID 1538 affiche un **PPID de 1**. Cela voudrait dire que le père du processus fils serait le processus **init**. En fait, au moment où le processus fils (1539) appelle la fonction **getppid()**, son processus père s'est déjà terminé, il devient donc **orphelin** et est "adopté" par le processus init de PID égal à 1.



Un processus **orphelin** est donc un processus dont le processus créateur est terminé. Tout processus orphelin est "**adopté**" par le processus **init** afin que celui-ci le supprime de la **table des processus** au moment où il se termine. Il s'agit d'un des services rendus par le processus **init**. Ceci est nécessaire pour que la table des processus ne se sature pas en processus zombi.

Test de la valeur de retour de fork()

Il est nécessaire de **tester la valeur de retour** de la fonction fork **immédiatement après** son exécution. Cela permet d'attribuer à chaque processus (père et fils) la portion du code qui lui est dédié. Le schéma général est le suivant :



Bien que les deux processus possèdent exactement le **même code**, on peut ainsi leur donner une **"direction"** d'exécution particulière.

Exemple 4.6 (fork - test de la valeur de retour). Soit le programme suivant

```

***** Test_fork_ret.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    pid_t id;

    id = fork();
    if (id == 0)
    {
        sleep(1);
        printf("Je suis le fils : PID=%d \t PPID=%d\n",getpid(),getppid());
        exit(0);
    }
    else
    {
        sleep(2);
        printf("Je suis le pere : PID=%d \t PPID=%d\n",getpid(),getppid());
        exit(0);
    }
    si je veux créer des autres processus, les faire ici
}

```

dont un exemple d'exécution fournit

```
# echo $$  
6705  
# ./Test_fork_ret  
Je suis le fils : PID=18891 PPID=18890  
Je suis le pere : PID=18890 PPID=6705  
#
```

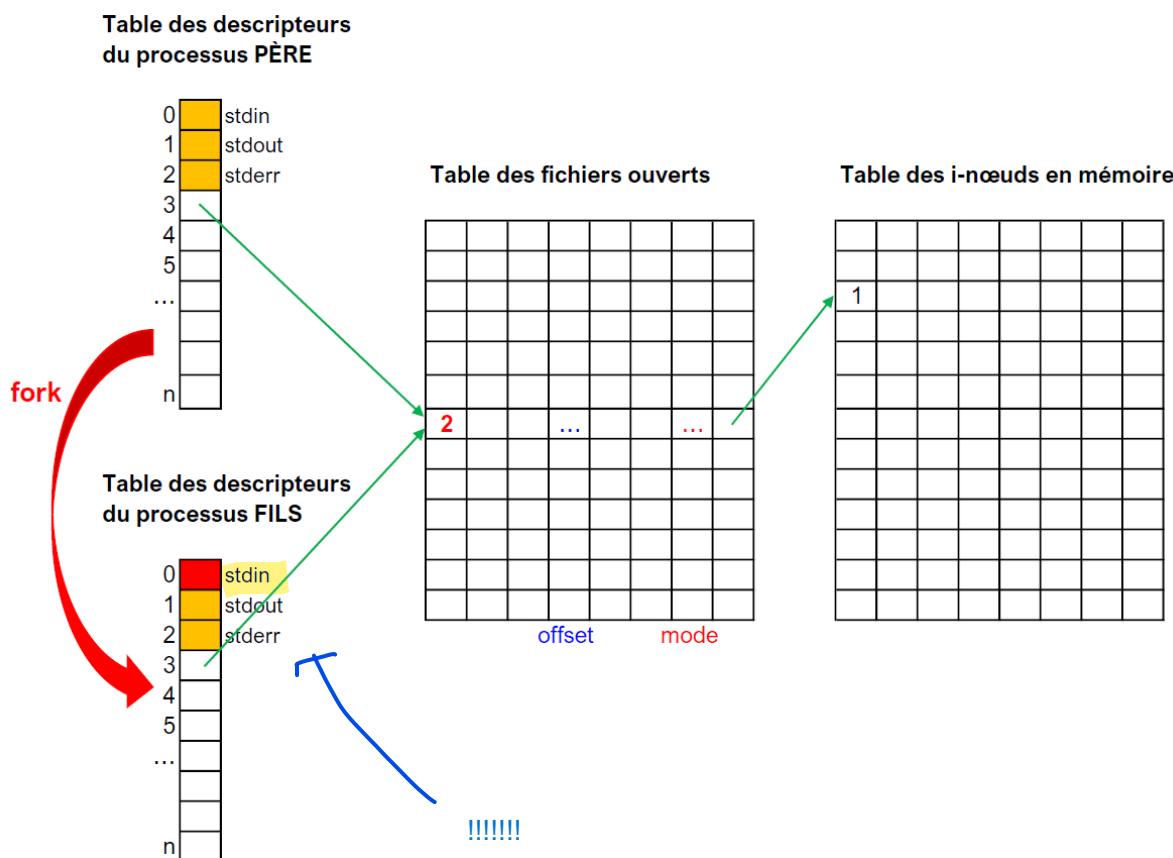
Remarquons que la présence des `sleep` assure simplement que le processus père vit toujours au moment où le processus fils exécute la fonction `getppid()`. Aucune obligation donc.



Héritage des fichiers ouverts par le père

Un processus fils hérite des fichiers ouverts par son processus père. Plus précisément, il hérite des **descripteurs des fichiers** ouverts par son père **avant l'appel de la fonction fork**. Ces descripteurs pointent alors vers les mêmes entrées dans la table des fichiers ouverts. Processus père et fils manipulent donc le **même offset** quand ils accèdent à un fichier partagé.

Voici le schéma de la situation pour un fichier ouvert par le processus père (via l'appel de la fonction `open` par exemple) et hérité par son processus fils :



La **duplication** du processus père par la fonction **fork** entraîne que

1. le **compteur** de l'entrée correspondante a été **incrémenté** dans la table des fichiers ouverts. Dans ce cas-ci, il est passé de 1 à 2.
2. l'entrée standard **stdin** du processus **fils** a été **fermée**. Il est en effet lancé **en arrière plan** et n'a plus accès au terminal pour réaliser des saisies claviers. Cela est nécessaire sinon il y aurait conflit entre processus père et fils s'ils souhaitaient réaliser tous deux une saisie clavier en même temps.

Exemple 4.7 (fork - héritage des fichiers ouverts). Soit le programme suivant

```
***** Test_fork_fic.cpp *****/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    pid_t idFils;
    int fd;
    int ret;
    char buffer[80];

    if ((fd = open("fichier.txt", O_RDONLY)) == -1)
    {
        perror("Erreur de open()");
        exit(1);
    }

    if ((idFils = fork()) == 0)
    {
        // Processus fils
        printf("\t(Fils) PID : %d\n", getpid());
        sleep(5);
        if ((ret = read(fd, buffer, 10)) == -1)
        {
            perror("(Fils) Erreur de read()");
            exit(1);
        }
        buffer[ret] = '\0';
        printf("\t(Fils) Lu : --%s--\n", buffer);

        sleep(10);
        close(fd);
        exit(0);
    }

    // Processus pere
    printf("(Pere) PID : %d\n", getpid());
    sleep(10);
    if ((ret = read(fd, buffer, 10)) == -1)
    {
        perror("(Pere) Erreur de read()");
        exit(1);
    }
    buffer[ret] = '\0';
    printf("(Pere) Lu : --%s--\n", buffer);
```

```

sleep(5);
close(fd);
exit(0);
}

```

dont un exemple d'exécution fournit

```

# cat fichier.txt
abcdefghijklmнопqrstuvwxyz
# ./Test_fork_fic
(Pere) PID : 26003
(Fils) PID : 26004
(Fils) Lu : --abcdefhij--
(Pere) Lu : --klmnopqrst--
#

```

Quelques remarques :

1. Processus père et fils ont lu **séquentiellement** dans le fichier en utilisant la “**même tête de lecture**”, le fils en premier et ensuite le père.
2. Bien qu'un seul appel à la fonction **open** ait été réalisé, chaque processus a dû fermer le fichier par un appel à la fonction **close**.
3. La présence de l'**exit(0)** dans le code exécuté par le processus fils est **nécessaire**, sans quoi, le processus fils exécuterait le code déjà exécuté par le père.

On peut se rendre également compte que les deux processus manipulent le **même fichier** avec la **même entrée** dans la table des fichiers ouverts et donc le **même offset**. Pour cela, il suffit d'utiliser la commande **lsof** dans un autre terminal :

```

# lsof -aod "3" | grep Test_fork
Test_fork 26003 student 3r REG 252,0 0t0 70755209 /home/student/fichier.txt
Test_fork 26004 student 3r REG 252,0 0t0 70755209 /home/student/fichier.txt
# lsof -aod "3" | grep Test_fork
Test_fork 26003 student 3r REG 252,0 0t10 70755209 /home/student/fichier.txt
Test_fork 26004 student 3r REG 252,0 0t10 70755209 /home/student/fichier.txt
# lsof -aod "3" | grep Test_fork
Test_fork 26003 student 3r REG 252,0 0t20 70755209 /home/student/fichier.txt
Test_fork 26004 student 3r REG 252,0 0t20 70755209 /home/student/fichier.txt
# lsof -aod "3" | grep Test_fork
#

```

Quelques remarques :

1. les **sleep** dans le code du programme permettent d'appeler la fonction **lsof** à des instants bien précis de l'exécution.
2. les paramètres de la commande **lsof** permettent d'afficher les fichiers ouverts de **descripteur 3** pour tous les processus dont le nom contient “**Test_fork**”.
3. le premier appel de **lsof** se fait entre 0 et 5 secondes après le lancement du programme. Père et fils n'ont pas encore réalisé de lecture et l'**offset est égal à 0**.
4. le second appel de **lsof** se fait entre 5 et 10 secondes après le lancement du programme. Le fils a lu 10 bytes dans le fichier mais on remarque que l'**offset est égal à 10** et est **identique pour les deux processus**.

5. le troisième appel de `lsof` se fait entre 10 et 15 secondes après le lancement du programme. Père et fils ont lu, chacun 10 bytes et on remarque que l'**offset est égal à 20** et est **identique pour les deux processus**.
6. le dernier appel de `lsof` se fait lorsque les deux processus se sont terminés.



Exemple 4.8 (fork - fermeture du stdin pour le fils). Soit le programme suivant

```
***** Test_fork_stdin.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    pid_t idFils;
    char buffer[80];

    if ((idFils = fork()) == 0)
    {
        // Processus fils
        printf("(Fils) PID : %d\n",getpid());
        printf("(Fils) PPID : %d\n",getppid());

        if (fgets(buffer,80,stdin) == NULL)
        {
            perror("(Fils) Erreur de fgets()");
            exit(1);
        }
        printf("(Fils) Lu : --%s--\n",buffer);

        exit(0);
    }

    // Processus pere
    printf("(Pere) PID : %d\n",getpid());
    printf("(Pere) je me termine.\n");
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./Test_fork_stdin
(Pere) PID : 777
(Pere) je me termine.
(Fils) PID  : 779
(Fils) PPID : 1
(Fils) Erreur de fgets(): Input/output error
#
```

On remarque bien que le processus fils a provoqué une **erreur** lors de l'appel à la fonction `fgets` en voulant réaliser une saisie clavier sur le `stdin`.



4.2.3 Terminer un processus : l'appel système exit

Un processus peut **terminer sa propre exécution** à tout moment en appelant la fonction **exit** et **retourner une valeur à son processus père**. Plus précisément, lors de l'appel de la fonction **exit**,

1. elle ferme tous les **descripteurs de fichiers** ouverts par le processus appelant,
2. tous les processus fils du processus appelant deviennent **orphelins** et sont adoptés par le processus **init**,
3. le signal **SIGCHLD** est envoyé au processus père du processus appelant,
4. une **valeur de retour comprise entre 0 et 255** est stockée dans la **table des processus**. Cette valeur pourra être récupérée par le processus **père** du processus appelant.
5. le processus appelant devient **zombi** après son exécution.

La documentation de la fonction **exit** peut être obtenue dans le man :

```
# man -s 3 exit
...
#
```

Le prototype de cette fonction est :

```
#include <stdlib.h>

void exit(int status);
```

Paramètre

Le paramètre **status** est la valeur entière que l'on **souhaite retourner** et stocker dans la **table des processus**. Cependant, étant donné que la valeur stockée dans la table des processus est une valeur entière comprise entre 0 et 255, la valeur qui sera réellement transmise correspond aux 8 bits de poids faible de la variable **status**, ou encore à **(status modulo 256)**.

Valeur de retour

Cette fonction n'a **aucun retour** vu que le processus devient zombi au terme de son exécution.

Exemple 4.9 (exit). Soit le programme suivant

```
***** Test_exit.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    int valeur = atoi(argv[1]);
    exit(valeur);
}
```

dont quelques exemples d'exécution fournissent

```
# ./Test_exit 17
# echo $?
17
# ./Test_exit 300
# echo $?
44
# ./Test_exit 255
# echo $?
255
# ./Test_exit 256
# echo $?
0
# ./Test_exit -10
# echo $?
246
#
```

On constate que :

1. Le programme **Test_exit** passe simplement en paramètre à la fonction **exit** la valeur du premier argument reçu en ligne de commande.
2. Le **shell** est le processus **père** du processus **Test_exit**. C'est donc lui qui **le supprime de la table des processus et récupère sa valeur de retour**. Il est possible d'interroger le shell et de lui demander d'afficher la dernière valeur de retour récupérée d'un de ses fils, c'est-à-dire le contenu de sa **variable d'environnement** **?**. Pour cela, on utilise la commande **echo \$?**.
3. Lorsque la valeur passée à **exit** est comprise entre 0 et 255, la valeur récupérée par le shell est identique.
4. Lorsque la valeur passée à **exit** n'est pas comprise entre 0 et 255, on voit que la valeur récupérée a été modifiée par un "modulo 256".



4.2.4 Attendre la fin d'un processus : l'appel système **wait**

La fonction **wait** permet à un processus appelant

valeur de retour !!! examen
pas de fils = -1
retour du wait : pid du FILS !!!

- de **supprimer** un de ses fils zombi de la **table des processus** (cas **non bloquant**),
- d'**attendre la fin d'un de ses processus fils** en cours d'exécution (cas **bloquant**).

Elle ne permet pas d'attendre la fin d'un processus fils particulier. Dans tous les cas, elle supprime le processus fils sélectionné de la table des processus et peut récupérer ou non la manière dont ce processus s'est terminé.

La documentation de la fonction **wait** peut être obtenue dans le man :

```
# man -s 2 wait
...
#
```

Le prototype de cette fonction est :

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

pid_t wait(int* status);
qui           comment
```

Paramètre

Le paramètre **status** est l'**adresse d'une variable** qui va recevoir toutes les **informations sur la fin du processus fils** terminé. Si l'on ne désire pas connaître ces informations, le paramètre **status** peut être égal à **NULL**.

Valeur de retour

Cette fonction retourne

- le **PID du processus fils qui s'est terminé** et qui a été **supprimé de la table des processus**.
- **-1** en cas d'erreur et, **errno** est positionné.

En cas d'erreur, les valeurs de **errno** les plus fréquentes sont

errno	signification
ECHILD	Le processus appelant n'a pas de fils.
EINTR	L'appel système a été interrompu par un signal
...	...

Algorithme

Pour être plus précis, l'algorithme de l'appel système **wait** est le suivant :

Algorithme 4.1 L'appel système wait.

```

Si le processus appelant ne possède pas de fils
  Alors
    Retour -1 et errno est positionné à ECHILD.
  Fin Si

Si le processus appelant possède au moins un fils zombi dans la table des processus (CAS NON BLOQUANT)
  Alors
    Choix d'un des fils zombi au hazard dans la table des processus.
    Suppression de ce fils zombi de la table des processus.
    Mise à jour de la variable pointée par status.
    Retour du PID du processus fils zombi en question.
  Fin Si

Si le processus appelant possède au moins un fils mais aucun fils zombi (CAS BLOQUANT)
  Alors
    Le processus appelant est bloqué jusqu'à
      Soit la fin d'un de ses processus fils. Dans ce cas :
        Suppression de ce fils de la table des processus.
        Mise à jour de la variable pointée par status.
        Retour du PID de ce processus fils.
      Soit la réception d'un signal. Dans ce cas :
        Retour -1 et errno est positionné à EINTR.
  Fin Si

```

On peut remarquer que la fonction **wait** n'est **pas nécessairement bloquante**. Si un processus fils zombi existe, la fonction **wait** n'est pas bloquante et retourne directement le PID de ce processus.

Exemple 4.10 (wait). Soit le programme suivant

```

***** Test_wait.cpp *****/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main()
{
  pid_t id1,id2,id3;
  pid_t id;

  if ((id1 = fork()) == -1)
  {
    perror("(Pere) Erreur de fork(1)");
    exit(1);
  }

  if (!id1)
  {
    // Code du fils 1
    printf("(Fils 1) Attente de 7 secondes (%d)\n",getpid());
    sleep(7);
    exit(0);
  }

  if ((id2 = fork()) == -1)

```

```

{
    perror("(Pere) Erreur de fork(2)");
    exit(1);
}

if (!id2)
{
    // Code du fils 2
    printf("(Fils 2) Attente de 3 secondes (%d)\n",getpid());
    sleep(3);
    exit(0);
}

if ((id3 = fork()) == -1)
{
    perror("(Pere) Erreur de fork(3)");
    exit(1);
}

if (!id3)
{
    // Code du fils 3
    printf("(Fils 3) Attente de 5 secondes (%d)\n",getpid());
    sleep(5);
    exit(0);
}

// Code du pere
while((id = wait(NULL)) != -1)      fait tout les fils
{
    printf("(Pere) Fils %d termine\n",id);
}

if (errno != ECHILD)
{
    perror("(Pere) Erreur de wait()");
    exit(1);
}

printf("(Pere) Plus de fils.\n");
exit(0);
}

```

dans lequel le processus père crée 3 processus fils et attend la fin de leur exécution. Un exemple d'exécution fournit

```

# ./Test_wait
(Fils 1) Attente de 7 secondes (19657)
(Fils 2) Attente de 3 secondes (19658)
(Fils 3) Attente de 5 secondes (19659)
(Pere) Fils 19658 termine
(Pere) Fils 19659 termine
(Pere) Fils 19657 termine
(Pere) Plus de fils.
#

```

On constate que :

1. Lors des 3 premiers appels de la fonction **wait**, celle-ci est **bloquante** et retourne le **PID**

du fils qui se termine.

2. Lors du dernier appel de la fonction **wait**, il n'y a plus de processus fils. Celle-ci n'est **pas bloquante**, retourne **-1** et **errno** est positionnée à **ECHILD**.
3. Le paramètre passé à la fonction **wait** est **NULL**. Cela signifie que le processus père ne récupère pas les informations concernant la fin de ses processus fils.



Récupération des informations sur la fin d'un processus fils

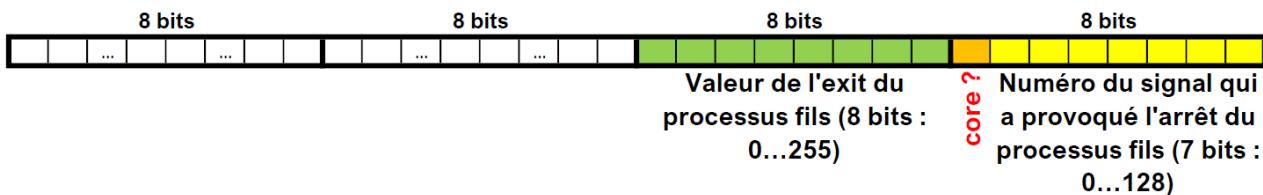
Un processus père peut récupérer toutes les informations sur la fin d'un de ses processus fils. Ces informations sont stockées dans la **table des processus** jusqu'au moment où le processus père exécute la fonction **wait** et que le fils zombi en question est sélectionné. Celui-ci est alors supprimé de la table des processus et toute l'information concernant sa fin est stockée dans un **entier** dont l'adresse est fournie en paramètre à la fonction **wait**.

Lors du retour de la fonction **wait** :

```
int id,status;
id = wait(&status);
```

1. la variable **id** contient le **PID du fils** terminé et qui vient d'être supprimé de la table des processus,
2. la variable **status** contient toute l'**information sur la fin** du processus fils en question.

La structure de la variable entière **status** est alors



L'information pertinente est stockée sur les **2 bytes** de poids faible de cette variable. Les 8 bits de droite (en jaune et orange) nous informent sur la manière dont le processus fils s'est terminé :

- **si les 8 bits sont à 0**, le processus s'est terminé suite à l'exécution de la fonction **exit** et la **valeur de l'exit** est stockée dans les 8 bits suivants (en vert).
- **si les 7 bits de droites** (en jaune) sont **non nuls**, ils correspondent au **numéro du signal** d'interruption qui a provoqué l'arrêt du processus.
- **si le 8ème bit** (en orange) est à **1**, cela signifie que le processus s'est terminé en provoquant la création d'un **fichier core**. Celui-ci est une **image mémoire** du processus au moment où celui-ci s'est arrêté. Ce fichier peut être analysé mais cela sort du cadre de ce cours.

Afin de récupérer ces informations, différentes **macros paramétrées** (dont le paramètre est la variable **status**) existent et sont à la disposition du programmeur dans le fichier sys/wait.h :

Macro	Utilité
<code>WIFEXITED(status)</code>	est vrai si le processus fils s'est terminé par un appel à <code>exit()</code> .
<code>→ WEXITSTATUS(status)</code>	fournit la valeur de l'exit du processus fils terminé.
<code>WIFSIGNALED(status)</code>	est vrai si le processus fils s'est terminé suite à la réception d'un signal .
<code>→ WTERMSIG(status)</code>	fournit le numéro du signal d'interruption qui a provoqué l'arrêt du processus fils.
<code>→ WCOREDUMP(status)</code>	signale si un fichier core a été créé.
...	...

Attention, les macros `WEXITSTATUS` et `WTERMSIG` n'ont de sens que si les macros `WIFxxx` correspondantes ont renvoyé une valeur vraie.

Exemple 4.11 (wait - récupération de la valeur de l'exit). Soit le programme suivant

```
***** Test_wait_exit.cpp *****/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main()
{
    pid_t idFils,id;
    int status;

    if ((idFils = fork()) == -1)
    {
        perror("(PERE) Erreur de fork()");
        exit(1);
    }

    if (idFils)
    {
        // Code du pere
        printf("(PERE) PID = %d\n",getpid());
        printf("(PERE) Attend la fin du fils...\n");

        if ((id = wait(&status)) == -1)
        {
            perror("(PERE) Erreur de wait()");
            exit(1);
        }

        printf("(PERE) status = 0x%x\n",status);

        if (WIFEXITED(status))
        {
            printf("(PERE) Le fils %d est termine par un exit(%d)\n",idFils,WEXITSTATUS(status));
        }
        exit(0);
    }

    // Code du fils
    printf("\t(FILS) PID = %d\n",getpid());
```

```

printf("\t(FILS) Attente de 5 secondes...\n");
sleep(5);
exit(7);
}

```

dont un exemple d'exécution fournit

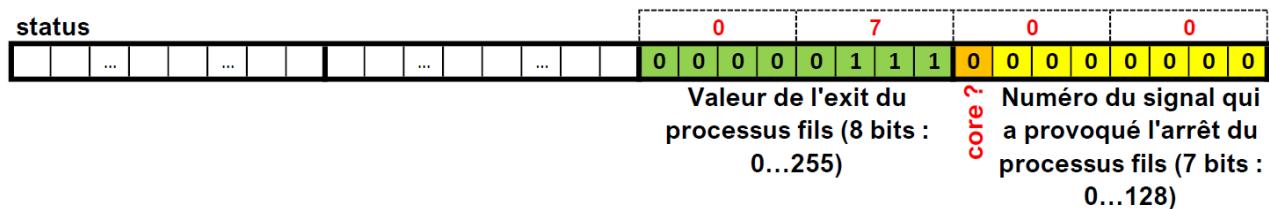
```

# ./Test_wait_exit
(PERE) PID = 26450
(PERE) Attend la fin du fils...
    (FILS) PID = 26451
    (FILS) Attente de 5 secondes...
(PERE) status = 0x700
(PERE) Le fils 26451 est terminé par un exit(7)
#

```

On constate que :

1. Le processus fils s'est terminé par un **exit(7)**.
2. Le processus père récupère l'information sur la fin de son fils 26451 dans sa variable **status** qui vaut, en hexadecimale, **0700**. Schématiquement, cela correspond à



les 8 bits de droite étant bien tous égaux à 0, indiquant que le fils s'est terminé par un appel à **exit()**.

3. La macro **WIFEXITED** a permis au père de savoir que son fils 26451 s'est terminé par un exit et la macro **WEXITSTATUS** lui a permis de récupérer la valeur (**7**) de cet exit.

Exemple 4.12 (wait - fin du fils à cause d'un signal). Soit le programme suivant

```

***** Test_wait_signal.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main()
{
    pid_t idFils,id;
    int status;

    if ((idFils = fork()) == -1)
    {
        perror("(PERE) Erreur de fork()");
        exit(1);
    }

```

```

if (idFils)
{
    // Code du pere
    printf("(PERE) PID = %d\n",getpid());
    sleep(2);
    printf("(PERE) Envoi du signal SIGINT au fils...\n");
    kill(idFils,SIGINT);      un kill ne tue pas un processus, il envoie un signal

    if ((id = wait(&status)) == -1)
    {
        perror("(PERE) Erreur de wait()");
        exit(1);
    }

    printf("(PERE) status = 0x%x\n",status);

    if (WIFSIGNALED(status))
    {
        printf("(PERE) Fils %d termine par reception du signal %d\n",idFils,WTERMSIG(status));
        if (WCOREDUMP(status)) printf("(PERE) Avec fichier core\n");
        else printf("(PERE) Sans fichier core\n");
    }
    exit(0);
}

// Code du fils
printf("\t(FILS) PID = %d\n",getpid());
printf("\t(FILS) Attente un signal...\n");
pause();
exit(7);
}

```

dont un exemple d'exécution fournit

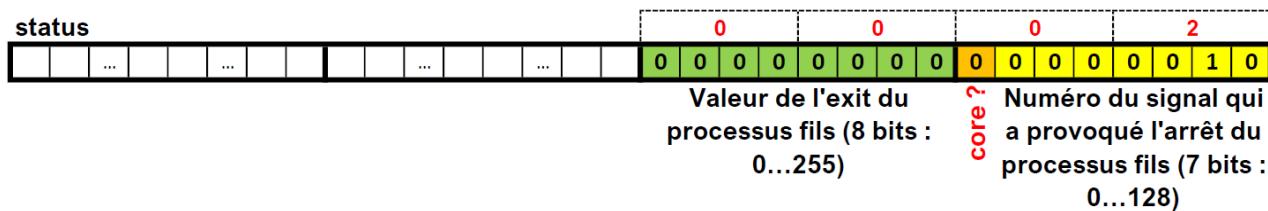
```

# ./Test_wait_signal
(PERE) PID = 30080
          (FILS) PID = 30081
          (FILS) Attente un signal...
(PERE) Envoi du signal SIGINT au fils...
(PERE) status = 0x2
(PERE) Fils 30081 termine par reception du signal 2
(PERE) Sans fichier core
#

```

On constate que :

1. En anticipant le chapitre sur les signaux, le processus père envoie le signal **SIGINT** (signal numéro **2**) à son processus fils à l'aide de la fonction **kill**, tandis que le processus fils se met en attente d'un signal à l'aide de la fonction **pause**.
2. La réception du signal **SIGINT** par le processus fils provoque son interruption et il ne se termine pas par **exit(7)** qu'il n'atteint pas. En effet, le comportement par défaut d'un processus ayant reçu ce signal est l'arrêt du processus (voir chapitre sur les signaux).
3. Le processus père récupère l'information sur la fin de son fils 30081 dans sa variable **status** qui vaut, en hexadécimal, **0002**. Schématiquement, cela correspond à



Les 7 bits de droite n'étant pas tous égaux à 0, cela indique au père que son fils s'est terminé à cause de la réception du signal numéro 2.

4. La macro `WIFSIGNALED` a permis au père de savoir que son fils 30081 s'est terminé à cause de la réception d'un signal et la macro `WTERMSIG` lui a permis de récupérer le numéro (2) de ce signal.
5. La macro `WCOREDUMP` a permis au père de savoir que son fils s'est terminé sans créer de fichier `core`.



Exemple 4.13 (wait - fin du fils à cause d'une erreur). Soit le programme suivant

```
***** Test_wait_core.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>

int main()
{
    pid_t idFils,id;
    int status;

    if ((idFils = fork()) == -1)
    {
        perror("(PERE) Erreur de fork()");
        exit(1);
    }

    if (idFils)
    {
        // Code du pere
        printf("(PERE) PID = %d\n",getpid());
        printf("(PERE) Attente de la fin du fils...\n");

        if ((id = wait(&status)) == -1)
        {
            perror("(PERE) Erreur de wait()");
            exit(1);
        }

        printf("(PERE) status = 0x%x\n",status);

        if (WIFSIGNALED(status))
        {
            printf("(PERE) Fils %d termine par reception du signal %d\n",idFils,WTERMSIG(status));
            if (WCOREDUMP(status)) printf("(PERE) Avec fichier core\n");
            else printf("(PERE) Sans fichier core\n");
        }
    }
}
```

```

    }
    exit(0);
}

// Code du fils
printf("\t(FILS) PID = %d\n",getpid());
printf("\t(FILS) Acces a une mauvaise adresse...\n");
char *buf = NULL;
strcpy(buf,"Hello !");
printf("\t(FILS) buf = --%s--\n",buf);
sleep(5);
exit(7);
}

```

dont un exemple d'exécution fournit

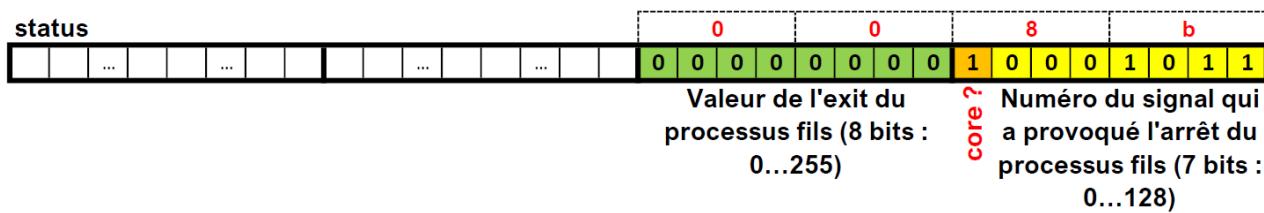
```

# ./Test_wait_core
(PERE) PID = 5261
(PERE) Attente de la fin du fils...
    (FILS) PID = 5262
    (FILS) Acces a une mauvaise adresse...
(PERE) status = 0x8b
(PERE) Fils 5262 termine par reception du signal 11
(PERE) Avec fichier core
#

```

On constate que :

1. Le processus fils tente d'accéder à une mauvaise adresse mémoire (**NULL**) lors de son appel à **strcpy**. Dès lors, le noyau du système d'exploitation lui envoie le signal **SIGSEGV** (signal numéro **11**), ce qui équivaut à un **segmentation fault**.
2. La réception du signal **SIGSEGV** par le processus fils provoque son interruption et il ne se termine pas par `exit(7)` qu'il n'atteint pas (ni le `sleep(5)` d'ailleurs). En effet, le comportement par défaut d'un processus ayant reçu ce signal est l'arrêt du processus et la création d'un fichier core (voir chapitre sur les signaux).
3. Le processus père récupère l'information sur la fin de son fils 5262 dans sa variable **status** qui vaut, en hexadecimal, **008b**. Schématiquement, cela correspond à



Les 7 bits de droite n'étant pas tous égaux à 0, cela indique au père que son fils s'est terminé à cause de la réception du signal numéro 11 (en binaire 1011).

4. La macro **WIFSIGNALED** a permis au père de savoir que son fils 5262 s'est terminé à cause de la réception d'un signal et la macro **WTERMSIG** lui a permis de récupérer le numéro (**11**) de ce signal.
5. La macro **WCOREDUMP** a permis au père de savoir que son fils s'est terminé en créant un fichier **core**.

4.2.5 Recouvrement d'un processus : les fonctions exec()

Au lieu d'exécuter une portion du code du processus qui l'a créé, il est possible pour un processus d'**exécuter un programme déjà compilé**. Ainsi, son code et ses données sont remplacés en mémoire par ceux du nouveau programme. On appelle cela le **recouvrement du processus** appellant.

Pour cela, nous disposons d'une **famille de fonctions "exec()**" dont voici quatre membres :

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

Notez qu'il n'existe aucune fonction qui s'appelle explicitement **exec**. Chacune de ces fonctions **remplace l'image mémoire du processus appelant** par l'image du nouveau programme reçu en paramètre. Il n'y a **pas ici de création de processus**. Avant et après recouvrement, le processus a conservé

- son PID et son PPID,
- ses propriétaires,
- la même priorité vis-à-vis du scheduler,
- le même répertoire de travail,
- le même temps d'attente pour la réception d'un signal SIGALRM (voir chapitre sur les signaux),
- les mêmes ensembles de signaux masqués et suspendus (voir chapitre sur les signaux),
- ...

Ces 4 fonctions effectuent exactement le même travail de recouvrement, la seule différence est la manière dont le programme à exécuter ainsi que ses arguments leur sont passés en paramètres.

Paramètres

Selon le cas, les paramètres sont

Paramètre	Signification
path	chemin complet vers le fichier exécutable à exécuter
file	nom de l'exécutable à exécuter, le chemin complet ne doit pas être précisé
arg, ...	liste de chaînes de caractères en nombre variables, correspondant aux mots situés sur la ligne de commande lors du lancement de l'exécutable. Le dernier paramètre doit être NULL.
argv	vecteur de chaînes de caractères correspondant aux mots situés sur la ligne de commande lors du lancement de l'exécutable. Le dernier élément de ce vecteur doit être NULL.

On peut donc classer ces fonctions en 3 ensembles selon que leur noms comportent le suffixe :

1. '**T'** : les mots sur la ligne de commande (pour lancer l'exécutable avec ses arguments) sont énumérés par une liste.
2. '**v'** : les mots sur la ligne de commande (pour lancer l'exécutable avec ses arguments) sont énumérés par un vecteur.
3. '**p'** : la recherche du fichier exécutable (dont le nom est précisé dans la variable **file**) se réalise grâce à la **variable d'environnement PATH** du processus. Sinon, le chemin complet (**path**) est précisé.

Valeur de retour

Ces fonctions

- n'ont **aucun retour** si le recouvrement s'effectue correctement,
- retournent la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EFAULT	Un des arguments pointe sur une mauvaise adresse.
EACCES	Le processus appelant n'a pas les droits suffisants sur l'exécutable et/ou sur le répertoire où il se trouve.
ENOENT	le fichier exécutable n'existe pas.
ENOMEM	Pas assez de mémoire.
...	...

Exemple 4.14 (exec). Soit le programme suivant

```
***** AffArg.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

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

```

printf("\tDebut de AffArg...\n");
printf("\tPID=%d PPID=%d UID=%d EUID=%d\n",getpid(),getppid(),getuid(),geteuid());

printf("\targc=%d\n",argc);
for (int i=0 ; i<argc ; i++)
    printf("\targv[%d]=%s\n",i,argv[i]);

printf("\tFin de AffArg.\n");
exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./AffArg aaa 11
    Debut de AffArg...
    PID=22771 PPID=12065 UID=1000 EUID=1000
    argc=3
    argv[0]=./AffArg
    argv[1]=aaa
    argv[2]=11
    Fin de AffArg.
#

```

Ce programme permet simplement d'afficher les identifiants du processus correspondant, ainsi que les différents "mots" de la ligne de commande, dont le nom de l'exéutable argv[0]. Considérons à présent le programme :

```

/***** Test_execl.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>

int main()
{
    printf("Debut de Test_execl...\n");
    printf("PID=%d PPID=%d UID=%d EUID=%d\n",getpid(),getppid(),getuid(),geteuid());

    if (execl("./AffArg","AffArg","abb","223",NULL) == -1)
    {
        perror("Erreur de execl()");
        exit(1);
    }

    printf("Fin de Test_execl.");
    exit(0);
}

```

dont un exemple d'exécution fournit

```
# ./Test_execl
Debut de Test_execl...
PID=23441 PPID=12065 UID=1000 EUID=1000
    Debut de AffArg...
    PID=23441 PPID=12065 UID=1000 EUID=1000
    argc=3
    argv[0]=AffArg
    argv[1]=abb
    argv[2]=223
    Fin de AffArg.
#
#
```

On constate que :

1. Avant et après appel de la fonction `execl`, le processus a gardé les **mêmes identifiants (PID et PPID) et propriétaires (UID et EUID)**, il s'agit donc bien du **même processus**. Aucun nouveau processus n'a été créé par l'appel de `execl`.
2. Dans les paramètres de la fonction `execl`, on a l'impression d'avoir passé 2 fois le nom de l'exécutable en paramètre. Il n'en est rien. Le premier paramètre est le **chemin d'accès vers le fichier exécutable** et le second paramètre est le **premier "mot" sur la ligne de commande**. Si, par exemple, on remplace le second paramètre "**AffArg**" par "**Prog**", un autre exemple d'exécution fournit :

```
# ./Test_execl
Debut de Test_execl...
PID=24611 PPID=12065 UID=1000 EUID=1000
    Debut de AffArg...
    PID=24611 PPID=12065 UID=1000 EUID=1000
    argc=3
    argv[0]=Prog
    argv[1]=abb
    argv[2]=223
    Fin de AffArg.
#
#
```

Le fichier exécutable et exécuté est toujours le même mais le premier mot sur la ligne de commande, récupéré dans `argv[0]` par `AffArg`, est bien "**Prog**".

3. Le programme **AffArg** est complètement exécuté. Par contre, dans le code de `Test_execl`, on remarque qu'il **ne va pas jusqu'au** dernier `printf("Fin de Test_execl.")`. En effet, `execl` n'est pas un appel de fonction conventionnel. Si tout se passe correctement, le processus est recouvert par le code de `AffArg` et **ne revient jamais au code de Test_execl**.



Exemple 4.15 (execv). Nous considérons ici le même programme **AffArg** que dans l'**exemple 4.14**. Soit le programme suivant

```
***** Test_execv.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <string.h>

int main()
{
```

```

printf("Debut de Test_execv...\n");
printf("PID=%d PPID=%d UID=%d EUID=%d\n",getpid(),getppid(),getuid(),geteuid());

char* mots[4];
mots[0] = (char*)malloc(7*sizeof(char)); strcpy(mots[0],"AffArg");
mots[1] = (char*)malloc(4*sizeof(char)); strcpy(mots[1],"abb");
mots[2] = (char*)malloc(4*sizeof(char)); strcpy(mots[2],"223");
mots[3] = NULL;

if (execv("./AffArg",mots) == -1)
{
    perror("Erreur de execv()");
    exit(1);
}

printf("Fin de Test_execv.");
exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./Test_execv
Debut de Test_execv...
PID=26845 PPID=12065 UID=1000 EUID=1000
    Debut de AffArg...
    PID=26845 PPID=12065 UID=1000 EUID=1000
    argc=3
    argv[0]=AffArg
    argv[1]=abb
    argv[2]=223
    Fin de AffArg.
#

```

On constate que tout se déroule exactement comme dans l'exemple précédent. La seule différence est la manière de passer en paramètres à la fonction `execv` les différents mots sur la ligne de commande.



Exemple 4.16 (execlp). Soit le programme suivant

```

***** Test_execlp.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>

int main()
{
    if (execlp("ls","ls","-l",NULL) == -1)
    {
        perror("Erreur de execlp()");
        exit(1);
    }

    exit(0);
}

```

dont un exemple d'exécution fournit

```
# ./Test_execlp
total 272
-rwxrwxr-x. 1 student student 8808 26 mai 17:36 AffArg
-rw-rw-r--. 1 student student 414 26 mai 17:36 AffArg.cpp
...
-rwxrwxr-x. 1 student student 8912 29 mai 11:12 Test_execl
-rw-rw-r--. 1 student student 425 29 mai 11:29 Test_execl.cpp
-rwxrwxr-x. 1 student student 8600 29 mai 14:20 Test_execlp
-rw-rw-r--. 1 student student 253 29 mai 14:20 Test_execlp.cpp
-rwxrwxr-x. 1 student student 8960 29 mai 11:41 Test_execv
-rw-rw-r--. 1 student student 661 29 mai 11:41 Test_execv.cpp
...
# echo $PATH
/usr/local/jdk1.8.0_111/bin:/opt/rh/devtoolset-6/root/usr/bin:/usr/lib64/qt-3.3/bin:
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:
/home/student/.local/bin:/home/student/bin
#
```

On constate que le premier paramètre de la fonction `execlp` est “**ls**” et que le chemin d'accès n'est pas spécifié. De plus, l'exécutable **ls** ne se trouve pas dans le répertoire courant mais bien dans le répertoire **/usr/bin**. Malgré tout, la fonction `execlp` trouve le fichier exécutable **ls** car elle utilise la **variable d'environnement PATH** qui contient le répertoire **/usr/bin**.

Si on remplaçait la fonction `execlp` par la fonction `execl` dans le code ci-dessus, un exemple d'exécution pourrait être

```
# ./Test_execlp
Erreur de execl(): No such file or directory
#
```

Dans ce cas, le fichier exécutable n'est pas trouvé car le chemin d'accès est inconnu.

4.2.6 Un exemple complet

Nous allons présenter ici un exemple faisant intervenir les 4 appels systèmes **fork**, **exec**, **exit** et **wait** :

- le processus père va interagir avec l'utilisateur afin de lui demander d'entrer, au clavier, deux entiers dont il souhaite calculer la somme.
- pour cela, le processus père va créer un processus fils qui sera chargé de réaliser ce calcul.
- Le processus fils va exécuter le programme somme qui reçoit en arguments 2 entiers et qui retourne, via l'appel de la fonction exit, la somme de ces deux entiers.
- le processus père va se **synchroniser** sur la fin du processus fils. En d'autres termes, le père va **attendre** que le fils se termine, récupérer le résultat fourni par le fils et afficher le résultat final en console avant de se terminer.

Exemple 4.17 (un exemple complet). Soit le programme suivant

```
***** somme.cpp ****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc,char* argv[])
{
    if (argc != 3)
    {
        printf("\t(somme) Erreur : Trop peu d'argument.\n");
        exit(1);
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int res = a + b;

    exit(res);
}
```

dont quelques exemples d'exécution fournissent

```
# ./somme 4
(somme) Erreur : Trop peu d'argument.
# ./somme 12 23
# echo $?
35
#
```

On constate que

1. le programme **somme** ne fait pas de saisie clavier, ni d'affichage en console. Il se comporte exactement comme une **commande** qui rend un **service**.
2. On affiche le résultat de son exécution en utilisant la commande **echo \$?**, résultat qui correspond à la valeur passée à la fonction **exit**.

3. les arguments reçus étant obligatoirement des **chaînes de caractères**, ceux-ci doivent être convertis (fonction **atoi**) en entiers afin de pouvoir réaliser le calcul.

Considérons à présent le programme :

```
***** Test_fork_exec.cpp *****
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main()
{
    pid_t id,idF;
    int status;
    int x,y,s;

    fflush(stdin);
    printf("Valeur de x = "); scanf("%d",&x);
    fflush(stdin);
    printf("Valeur de y = "); scanf("%d",&y);

    if ((id = fork()) == -1)
    {
        perror("Erreur de fork()");
        exit(1);
    }

    if (id == 0)
    {
        // Processus fils
        char bufX[10],bufY[10];
        sprintf(bufX,"%d",x);
        sprintf(bufY,"%d",y);

        if (execl("./somme","somme",bufX,bufY,NULL) == -1)
        {
            perror("Erreur de exelp()");
            exit(1);
        }
    }

    // Processus pere
    if ((idF = wait(&status)) == -1)
    {
        perror("Erreur de wait()");
        exit(2);
    }

    if (idF != id)
    {
        printf("Pas le bon fils\n");
        exit(3);
    }

    if (!WIFEXITED(status))
    {
        printf("Fils non termine par exit\n");
    }
}
```

```

    exit(4);
}

s = WEXITSTATUS(status);
printf("Somme de x et y = %d\n",s);

exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./Test_fork_exec
Valeur de x = 6
Valeur de y = 15
Somme de x et y = 21
# ./Test_fork_exec
Valeur de x = 300
Valeur de y = 200
Somme de x et y = 244
#

```

On constate que :

1. Afin de passer correctement les arguments au programme somme lors de l'`exec`, le processus fils doit **convertir** les deux entiers en **chaînes de caractères**. Cela est nécessaire vu que l'on ne peut passer que des chaînes de caractères en arguments à un programme.
2. Le processus père **attend** la fin du fils. Il vérifie ensuite l'identité du fils terminé (il aurait pu en lancer plusieurs) et que celui-ci s'est bien terminé par un `exit`. Si c'est le cas, le résultat du calcul est récupéré grâce à la macro `WEXITSTATUS`.
3. Si le résultat de la somme est supérieur à 255 (ou inférieur à 0), celui-ci est erroné. C'est normal étant donné que la fonction `exit` ne peut retourner, au processus père, qu'une valeur comprise entre 0 et 255.



Et si on voulait aller plus loin ?

- On pourrait imaginer que le temps d'exécution de somme soit plus long (pour d'autres calculs plus compliqués qu'une simple somme) et lancer, **en parallèle, plusieurs processus fils**. Pendant ce temps, le processus père pourrait s'occuper d'autres choses (dialoguer avec l'utilisateur par exemple). Le père devrait alors être prévenu qu'un fils a fini son travail et récupérer son résultat. Cela ne peut se faire que de **manière asynchrone**, c'est-à-dire que le père ne reste pas en permanence bloqué sur la fonction `wait`. Il doit alors être prévenu et cela ne peut se faire que par l'intermédiaire de l'étude des **signaux d'interruption**.
- Dans l'exemple ci-dessus, le fils ne peut retourner qu'une valeur entière comprise entre 0 et 255. Si l'on voulait que le fils communique au père un résultat plus grand ou même un résultat ayant un autre format comme une chaîne de caractères ou une structure, la fonction `exit` en serait incapable. Cela ne peut se faire que par un mécanisme plus élaboré de **communication entre processus**. Celui-ci sera abordé plus tard et porte le nom de "**Inter Process Communication**" (IPC).

Chapitre 5

Les signaux

5.1 Introduction

5.1.1 Généralités

Les **signaux** sous Linux constituent ce que l'on appelle des **interruptions logicielles**. Il s'agit d'**événements asynchrones** qui peuvent donc se produire à n'importe quel moment pendant l'exécution d'un processus. Un processus ne peut pas prédire s'il va recevoir un signal, ni même savoir quand il va en recevoir un. C'est ce que l'on appelle l'**asynchronisme** en programmation multi-processus.

L'événement associé à un signal peut être

– **externe au processus** :

- frappe du clavier (par exemple un <CTRL-C> provoque l'envoi du signal SIGINT),
- signal émis par un utilisateur à l'aide de la commande `kill`,
- signal émis par un autre processus à l'aide de la primitive `kill`,
- signal d'horloge (signal SIGALRM),
- signal émis par un fils à son père lorsqu'il se termine (signal SIGCHLD),
- ...

– **interne au processus** :

- erreur virgule flottante (signal SIGFPE),
- accès à une adresse mémoire hors de l'espace adressable (signal SIGSEGV - SEGV pour Segment Violation),
- détection d'une instruction illégale (signal SIGILL),
- ...

Il existe de nombreux signaux ayant chacun leur signification particulière. Ceux-ci sont représentés sur le système par un **entier positif** mais également par une **macro** associée. Par exemple, le signal SIGINT est le signal numéro 2 et SIGINT est une macro valant 2. Pour s'en rendre compte, il suffit d'afficher le contenu du fichier `/usr/include/bits/signum.h` :

```
# cat /usr/include/bits/signum.h
...
/* Signals. */
#define SIGHUP      1 /* Hangup (POSIX). */
#define SIGINT      2 /* Interrupt (ANSI). */
#define SIGQUIT      3 /* Quit (POSIX). */
#define SIGILL      4 /* Illegal instruction (ANSI). */
#define SIGTRAP      5 /* Trace trap (POSIX). */
#define SIGABRT      6 /* Abort (ANSI). */
#define SIGIOT      6 /* IOT trap (4.2 BSD). */
#define SIGBUS      7 /* BUS error (4.2 BSD). */
#define SIGFPE      8 /* Floating-point exception (ANSI). */
#define SIGKILL      9 /* Kill, unblockable (POSIX). */
#define SIGUSR1     10 /* User-defined signal 1 (POSIX). */
#define SIGSEGV     11 /* Segmentation violation (ANSI). */
#define SIGUSR2     12 /* User-defined signal 2 (POSIX). */
#define SIGPIPE     13 /* Broken pipe (POSIX). */
#define SIGALRM     14 /* Alarm clock (POSIX). */
#define SIGTERM     15 /* Termination (ANSI). */
#define SIGSTKFLT    16 /* Stack fault. */
#define SIGCLD      SIGCHLD /* Same as SIGCHLD (System V). */
#define SIGCHLD     17 /* Child status has changed (POSIX). */
#define SIGCONT     18 /* Continue (POSIX). */
#define SIGSTOP     19 /* Stop, unblockable (POSIX). */
...
#define _NSIG      65 /* Biggest signal number + 1
(including real-time signals). */
...
#
```

La commande `kill` avec l'option `-l` permet également d'obtenir la liste des signaux existants :

```
# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT      7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
#
```

Le nombre de signaux est limité par la macro `NSIG` qui est une redéfinition de la macro `_NSIG` rencontrée ci-dessus :

```
# cat /usr/include/signal.h
...
#define NSIG _NSIG
...
#
```

5.1.2 Comportement par défaut vis à vis des signaux

Lorsqu'un processus reçoit un signal, il réagit en conséquence. Si rien n'est programmé en ce sens, un processus dispose d'un **comportement par défaut** pour chacun des signaux. Plus précisément, une action est attribuée par défaut à chaque signal. Pour s'en rendre compte, il suffit de consulter le man :

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	2	Term	Interruption depuis le clavier.
SIGQUIT	3	Core	Demande « Quitter » depuis le clavier.
SIGILL	4	Core	Instruction illégale.
SIGABRT	6	Core	Signal d'arrêt depuis abort(3).
SIGFPE	8	Core	Erreur mathématique virgule flottante.
SIGKILL	9	Term	Signal « KILL ».
SIGSEGV	11	Core	Référence mémoire invalide.
SIGPIPE	13	Term	Écriture dans un tube sans lecteur.
SIGALRM	14	Term	Temporisation alarm(2) écoulée.
SIGTERM	15	Term	Signal de fin.
SIGUSR1	30,10,16	Term	Signal utilisateur 1.
SIGUSR2	31,12,17	Term	Signal utilisateur 2.
SIGCHLD	20,17,18	Ign	Fils arrêté ou terminé.
SIGCONT	19,18,25	Cont	Continuer si arrêté.
SIGSTOP	17,19,23	Stop	Arrêt du processus.
...			
Les signaux SIGKILL et SIGSTOP ne peuvent ni capturés ni ignorés.			
...			
#			

Comme on le voit, il existe 5 **comportements par défaut** différents :

1. **Term** : le processus se termine
2. **Ign** : le processus ignore le signal qui n'a donc aucun effet
3. **Core** : le processus se termine avec la création d'un fichier core (image mémoire du processus au moment de la terminaison)
4. **Stop** : le processus est suspendu mais ne se termine pas. Il est bloqué jusqu'à la réception du signal SIGCONT.
5. **Cont** : le processus reprend son exécution à l'endroit où il était suspendu.

Il s'agit en quelque sorte d'une "**table de correspondance**" entre "signal reçu" et "action à exécuter en cas de réception de ce signal". Par exemples,

- le comportement par défaut d'un processus qui reçoit le signal SIGINT est de se terminer.
- le comportement par défaut d'un processus qui reçoit le signal SIGCHLD est de l'ignorer.
- ...

La prise en compte d'un signal par un processus entraîne donc l'exécution d'une fonction spécifique appelée **handler de signal**. Par défaut, il existe donc 5 handlers de signaux **Term**, **Ign**, **Core**, **Stop** et **Cont** qui sont donc prédéfinis dans le système. Par la suite, nous verrons qu'il est possible de remplacer ces comportements par défaut, et de là installer ses propres handlers de signaux, par des fonctions choisies par le programmeur pour réaliser un traitement spécifique.

5.1.3 Envoi d'un signal : la commande kill

Un utilisateur ne peut envoyer un signal qu'à un processus dont il est **propriétaire**, sinon l'envoi du signal n'aura aucun effet. Pour cela, il peut utiliser la commande **kill** dont la documentation complète peut être obtenue dans le man :

```
# man kill
...
#
```

Cette commande dispose de plusieurs options mais l'utilisation générale est la suivante :

```
kill [-signal] pid
```

où

- **pid** est le PID du processus à qui l'on souhaite envoyer un signal.
- **[-signal]** est optionnel. S'il est n'est pas utilisé, le signal envoyé par défaut est SIGTERM.
- **signal** peut être soit le **numéro du signal** à envoyer, soit le **nom du signal**. Pour des raisons de portabilité, il est recommandé d'utiliser plutôt le nom du signal que son numéro.

Exemple 5.1 (kill). Soit le programme suivant

```
***** MiseEnPause.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Avant pause()\n");
    pause();
    printf("Après pause()\n");
    exit(0);
}
```

dont un premier exemple d'exécution fournit

```
# ./MiseEnPause &
[1] 16317
Avant pause()
# ps
  PID TTY          TIME CMD
16012 pts/3    00:00:00 bash
16317 pts/3    00:00:00 MiseEnPause
16330 pts/3    00:00:00 ps
# kill 16317    par défaut ça envoie le sigterm
[1]+ Complété ./MiseEnPause
# ps
  PID TTY          TIME CMD
16012 pts/3    00:00:00 bash
16355 pts/3    00:00:00 ps
#
```

Quelques constatations :

1. En anticipant un peu la suite du chapitre, la fonction `pause` est **bloquante**, elle met le processus en pause jusqu'à la réception d'un signal.
2. la commande `kill` a été utilisée sans option, le signal **SIGTERM** a été envoyé au processus dont le PID est 16317. Par défaut, le comportement vis-à-vis de SIGTERM est la fin immédiate du processus. On remarque que le processus n'a pas exécuté le second `printf` et n'a donc jamais atteint `exit(0)`.

Un autre exemple d'exécution est le suivant :

```
# ./MiseEnPause &
[1] 20045
Avant pause()
# kill -2 20045
[1]+ Interrompre ./MiseEnPause
# ./MiseEnPause &
[1] 20076
Avant pause()
# kill -SIGINT 20076
[1]+ Interrompre ./MiseEnPause
#
```

La commande `kill` a ici été utilisée avec l'option précisant le signal que l'on désire envoyer, soit par son numéro (`-2`), soit par son nom (`-SIGINT`). Dans les deux cas, le comportement par défaut du signal SIGINT est la fin du processus.

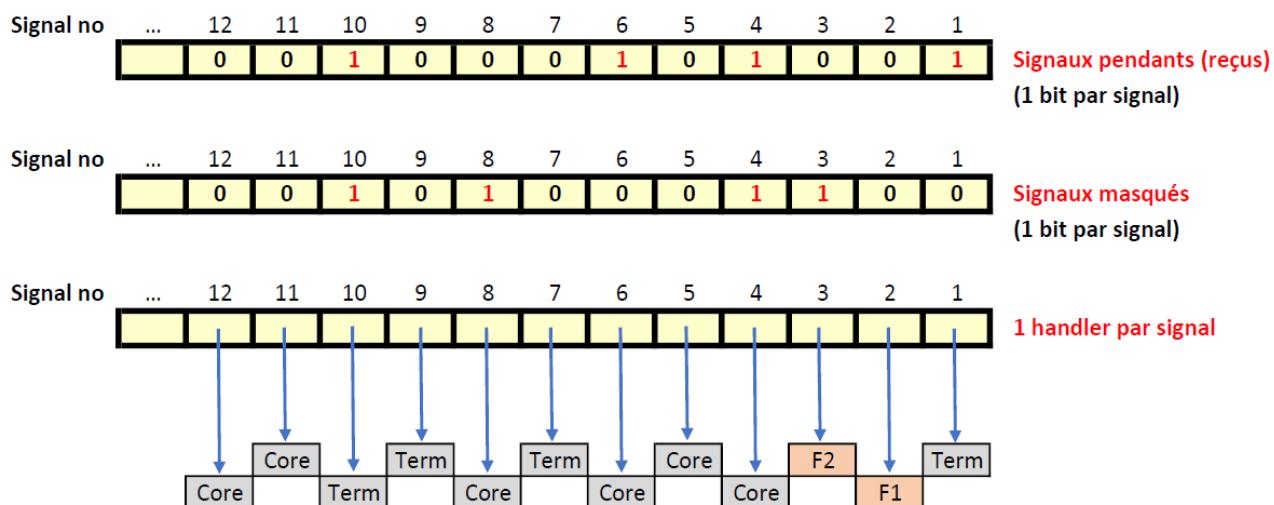


5.1.4 Gestion et traitement des signaux reçus

Afin de gérer la réception et le traitement des signaux reçus, un processus dispose de 3 vecteurs (de taille `NSIG`) :

1. le vecteur des **signaux reçus** (ou pendants),
2. le vecteur des **signaux masqués**,
3. le vecteur des **handlers de signaux**.

Les deux premiers vecteurs sont des vecteurs de bits où chaque case est associée à un signal particulier. De même, chacune des cases du dernier vecteur est spécifique à un signal mais celles-ci contiennent un pointeur vers la fonction handler correspondante. Schématiquement, cela correspond à la situation suivante :



qui est un exemple de l'état d'un processus vis-à-vis des signaux à un moment donné.

Le **vecteur des signaux reçus** mémorise les signaux qui ont été reçus par le processus. Etant donné qu'il s'agit d'un vecteur de bits, il est incapable de mémoriser le nombre de fois qu'un même signal a été reçu. Dès qu'un signal est reçu par un processus, ce vecteur est mis à jour.

Le **vecteur des signaux masqués** est l'ensemble des signaux qu'un processus a décidé de ne pas traiter. Ce vecteur n'empêche pas un processus de recevoir un signal (le vecteur des signaux reçus est toujours mis à jour) mais celui-ci n'entrainera pas l'exécution du handler correspondant.

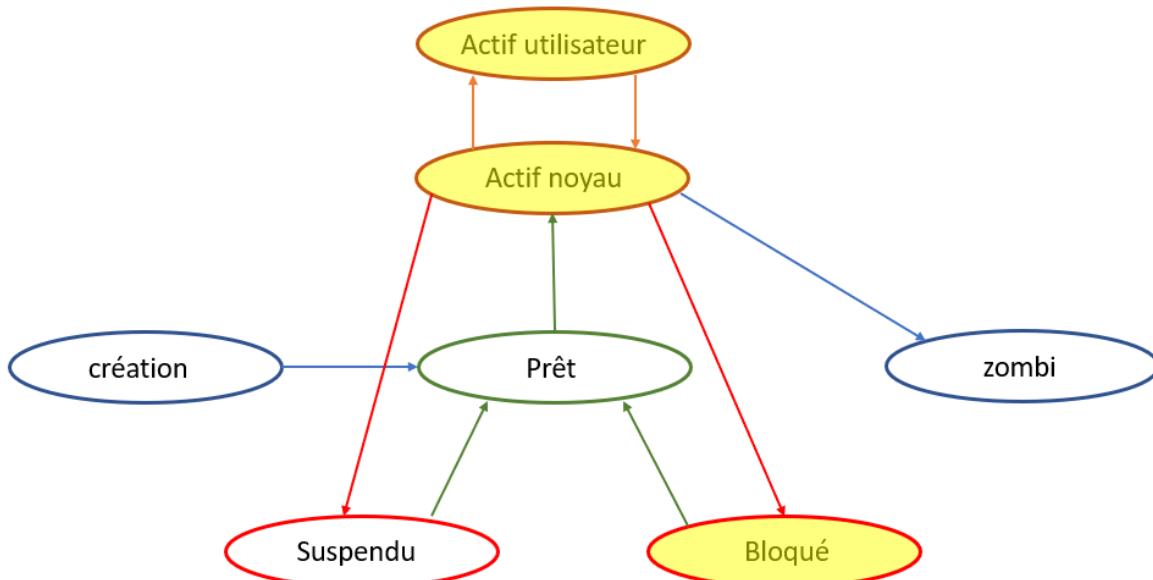
Le **vecteur des handlers** est l'ensemble des fonctions qui seront potentiellement exécutées en cas de réception d'un signal, potentiellement car si un signal est masqué, son handler ne sera pas exécuté en cas de réception de celui-ci.

Selon l'état de ces vecteurs, un signal peut être

- **pendant** : cela signifie que ce signal a été reçu, n'est pas masqué (c'est-à-dire n'apparaît pas dans le vecteur des signaux masqués) mais n'a pas encore été traité par le processus. Il est donc en attente de traitement. Sur le schéma ci-dessus, les signaux **1** et **6** sont pendants.
- **masqué** : cela signifie que ce signal n'a pas (encore) été reçu mais qu'il apparaît dans le vecteur des signaux masqués. Si le processus reçoit ce signal, celui-ci n'aura aucun effet. Sur le schéma ci-dessus, les signaux **3** et **8** sont masqués.
- **bloqué** : il s'agit d'un signal qui a été reçu par le processus et qui est actuellement masqué. Cela ne signifie pas qu'il ne sera jamais traité par le processus. Si celui-ci décide de le démasquer (remettre 0 dans la case correspondante du vecteur des signaux masqués), ce signal sera alors traité par l'exécution du handler correspondant. Sur le schéma ci-dessus, les signaux **4** et **10** sont bloqués.
- **délivré** : il s'agit d'un signal non masqué, qui a été reçu par le processus et dont le handler correspondant a été exécuté. Sur le schéma ci-dessus, le signal **2** pourrait avoir été délivré. Remarquez que le bit correspondant dans le vecteur des signaux reçus est remis à 0 avant l'exécution du handler.
- **ignoré** : il s'agit d'un signal dont le handler est **Ign**. Dans ce cas, si le signal est reçu par le processus, le bit correspondant dans le vecteur des signaux reçus est remis à 0 mais aucun traitement n'est effectué pour lui.

Remarquons finalement sur le schéma ci-dessus que les handlers des signaux **2** et **3** ne sont plus les handlers par défaut mais bien des fonctions **F1** et **F2** mise en place par le programmeur (voir plus loin).

Selon l'état du processus, celui-ci va réagir différemment à la réception d'un signal. Rapelons ici le schéma du cycle de vie d'un processus :

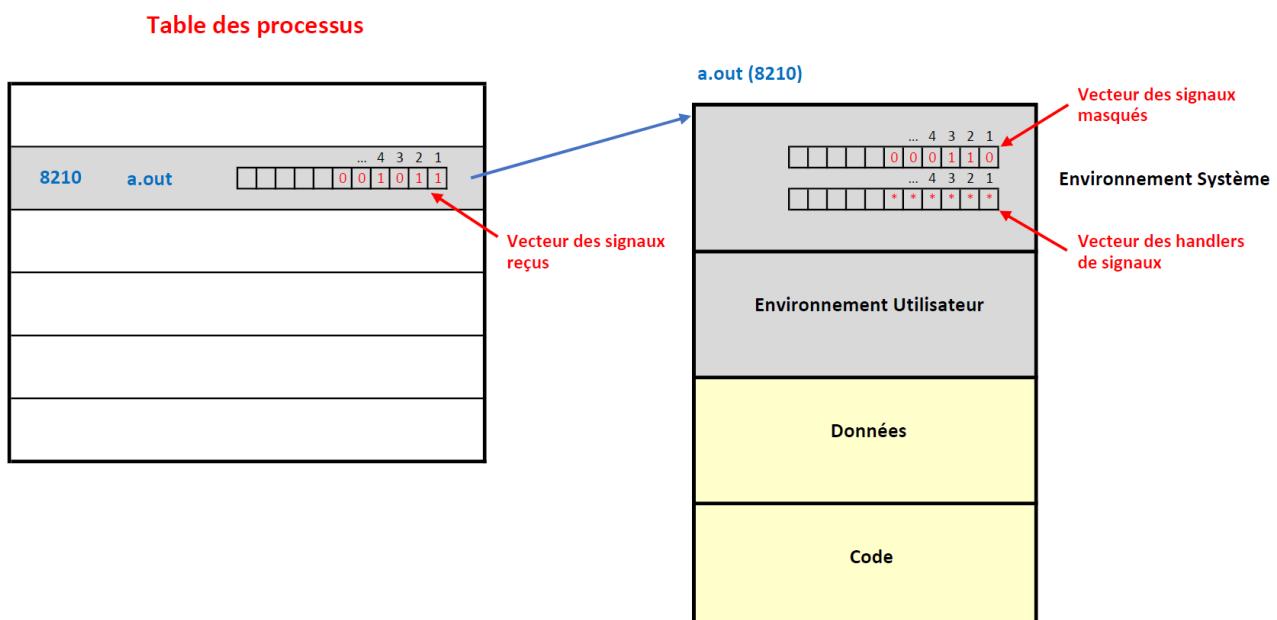


Trois cas de figure se présentent :

1. Le processus est en **mode utilisateur**, c'est-à-dire qu'il exécute des lignes de code programmées par l'utilisateur. Dans ce cas,
 - (a) Le processus scrute le vecteur des signaux reçus entre chaque ligne de code exécutée.
 - (b) Pour chaque signal pendant détecté (c'est-à-dire reçu et non masqué), il remet le bit à 0 dans le vecteur des signaux reçus avant d'exécuter le handler correspondant.
 - (c) Le processus retourne ensuite à l'endroit (entre les deux lignes de code) où il a été interrompu et poursuit son exécution.
2. Le processus est en **mode noyau**, c'est-à-dire qu'il exécute le code d'un appel système (code non programmé par l'utilisateur). Dans ce cas, rien ne se passe vis-à-vis des signaux. Ce n'est que lorsque l'appel système sera terminé que le processus va repasser en mode utilisateur et recommencer à scruter le vecteur des signaux reçus.
3. Le processus est **bloqué**, c'est-à-dire **endormi, sur un appel système**. Dans ce cas, si le signal reçu n'est pas masqué, le processus
 - (a) remet à 0 le bit correspondant dans le vecteur des signaux reçus,
 - (b) exécute le handler correspondant,
 - (c) retourne **juste après l'appel système** sur lequel il était endormi. Le **retour** de l'appel système est alors **-1** et **errno a été positionné à EINTR**. On dit alors que l'appel système a été interrompu.

Bien sûr, si le handler exécuté correspond à la fin du processus, celui-ci ne revient jamais de son interruption et devient alors zombi.

Les 3 vecteurs utilisés pour la gestion des signaux par un processus se situent à différents endroits en mémoire. Le schéma suivant illustre la situation pour un processus a.out de PID égal à 8210 :



On remarque que

- la **vecteur des signaux reçus** se situe au niveau de la table des processus. La commande **kill**, qui permet d'envoyer un signal à un processus, est donc un appel système qui modifie une des tables systèmes. Elle positionne simplement le bit à 1 à la case correspondante dans la table des processus.
- les **vecteurs des signaux masqués** et des **handlers de signaux** se situent au niveau de "l'environnement système" du processus. Ce ne sont donc pas des variables déclarées explicitement par l'utilisateur (zone "Données" du processus).

Le programmeur/utilisateur ne peut donc pas directement modifier le vecteur des signaux masqués et des handlers de signaux, il doit pour cela utiliser les fonctions prévues à cet effet (voir plus loin).

Plus précisément, il sera possible de

1. **armer un signal** : il s'agit de modifier le vecteur des handlers pour ce signal et donc de lui attribuer un handler particulier. Pour cela, il faut utiliser la fonction **sigaction** décrite plus loin.
2. **masquer un signal** : il s'agit de modifier le vecteur des signaux masqués pour ce signal en positionnant à 1 la case correspondante. Pour cela, il faut utiliser la fonction **sigprocmask** décrite plus loin.

5.2 Les ensembles de signaux : le type sigset_t

Afin de pouvoir masquer des signaux et manipuler des vecteurs de signaux, un type de données particulier (plus précisément une structure) existe. Il s'agit du type **sigset_t** défini dans `/usr/include/signal.h` qui représente un ensemble de signaux, comme le vecteur des signaux masqués par exemple :



D'un point de vue pratique, il s'agit d'un vecteur de bits dans lequel un signal peut être présent une seule fois (bit à 1 à l'indice correspondant au numéro du signal) ou pas du tout (bit à 0 à l'indice correspondant au numéro du signal). Il ne peut donc pas contenir plusieurs occurrences du même signal.

Un ensemble de signaux du type **sigset_t** peut être manipulé à l'aide des fonctions suivantes :

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

où

- **sigemptyset** permet d'initialiser le vecteur set avec tous des 0. L'ensemble obtenu est donc vide.
- **sigfillset** permet d'initialiser le vecteur set avec tous des 1. L'ensemble obtenu contient donc tous les signaux existants.
- **sigaddset** et **sigdelset** permettent respectivement d'ajouter et de retirer le signal signum de l'ensemble set,
- **sigismember** retourne 1 si le signal signum se trouve dans l'ensemble set (c'est-à-dire si le bit correspondant est à 1), et 0 dans le cas contraire.

Valeurs de retour

Ces fonctions retournent

- la valeur 0 en cas de succès (1 ou 0 pour sigismember),
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EINVAL	le numéro de signal signum n'est pas valide.
...	...

Exemple 5.2 (sigset_t). Soit le programme suivant

```
***** TestSigset_t.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void afficheMasque(const sigset_t* masque,int val);

int main()
{
    sigset_t masque;

    printf("Masque non initialise : "); afficheMasque(&masque,20);

    sigemptyset(&masque);
    printf("Masque initialise a 0 : "); afficheMasque(&masque,20);

    sigaddset(&masque,SIGINT);
    printf("Ajout de SIGINT (2) : "); afficheMasque(&masque,20);

    sigaddset(&masque,SIGCHLD);
    printf("Ajout de SIGCHLD (17) : "); afficheMasque(&masque,20);

    sigaddset(&masque,SIGSEGV);
    printf("Ajout de SIGSEGV (11) : "); afficheMasque(&masque,20);
```

```

sigdelset(&masque,SIGINT);
printf("Retrait de SIGINT (2) : "); afficheMasque(&masque,20);

sigfillset(&masque);
printf("Masque initialise a 1 : "); afficheMasque(&masque,20);

sigdelset(&masque,SIGPIPE);
printf("Retrait de SIGPIPE (13) : "); afficheMasque(&masque,20);

exit(0);
}

void afficheMasque(const sigset_t* masque,int val)
{
    for (int sig=val ; sig>0 ; sig--) printf("%d",sigismember(masque,sig));
    printf("\n");
}

```

dont un exemple d'exécution fournit

```

# ./TestSigset_t
Masque non initialise : 11110100111010010000
Masque initialise a 0 : 00000000000000000000
Ajout de SIGINT (2) : 0000000000000000000010
Ajout de SIGCHLD (17) : 0001000000000000000010
Ajout de SIGSEGV (11) : 0001000001000000000010
Retrait de SIGINT (2) : 00010000010000000000
Masque initialise a 1 : 11111111111111111111
Retrait de SIGPIPE (13) : 11111110111111111111
#

```

On constate que :

1. Comme toute variable non initialisée, la variable **masque** contient n'importe quoi avant son initialisation par la fonction **sigemptyset**.
2. La fonction **sigismember** retournant 1 ou 0 a permis d'écrire la fonction d'affichage d'une variable de type **sigset_t**. Le paramètre val de cette fonction correspond au nombre de signaux que l'on désire afficher (du numéro 1 au numéro val de la droite vers la gauche). On affiche donc ici uniquement les 20 premiers signaux.



5.3 Armement d'un signal : la fonction sigaction

Pour rappel, **armer un signal** signifie

1. **remplacer le comportement** précédent (ou par défaut) d'un processus vis-à-vis d'un signal, c'est-à-dire
2. associer à un numéro de signal une fonction (appelée **handler de signal**) qui sera, si ce signal n'est pas masqué, exécutée en cas de réception du signal,
3. **modifier le vecteur des handlers** à la case correspondante au signal à **armer**, et faire pointer cette case vers le code de nouvelle fonction handler.

La fonction à utiliser afin d'armer un signal est la fonction suivante :

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

où

- **signum** est le numéro du signal que l'on désire armer,
- **act** est une structure (de type **struct sigaction** expliquée en détails ci-dessous) qui contient le nouvel handler à installer,
- **oldact** est une structure (de type **struct sigaction** également) contenant l'ancien comportement (c'est-à-dire l'ancien handler) du processus vis-à-vis du signal signum. Celui-ci peut être NULL si on ne souhaite pas récupérer l'ancien comportement.

Cette fonction met donc en place le nouvel handler de signal au niveau du vecteur de handlers de signaux du processus et permet de récupérer l'ancien comportement du processus vis-à-vis du signal (à des fins, par exemple, de restauration ultérieure du comportement précédent).

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EINVAL	soit le numéro de signal signum n'est pas valide, soit signum correspond à SIGKILL ou SIGSTOP
EFAULT	act ou oldact correspondent à des adresses invalides
...	...

On remarque qu'il est **impossible d'armer les signaux SIGKILL (9) et SIGSTOP (19)**. Cela permet de garder quoi qu'il arrive un accès au processus via une interruption.

5.3.1 La structure sigaction

La structure sigaction est définie par

```
#include <signal.h>

struct sigaction
{
    void     (*sa_handler)(int);
    sigset_t sa_mask;
    int      sa_flags;
};
```

où

- **sa_handler** est un **pointeur vers la fonction handler** à installer. En pratique, il s'agit simplement du nom de la fonction que l'on désire installer comme handler.
- **sa_mask** correspond à l'**ensemble de signaux qui seront masqués pendant l'exécution du handler** (en plus de celui en cours de traitement qui est masqué automatiquement).
- **sa_flags** est un paramètre permettant de modifier l'armement “standard” du signal (voir plus loin).

Le champ principal de cette structure est évidemment **sa_handler**. Les deux autres champs permettent d'affiner la configuration de l'armement du signal. Notez néanmoins qu'il est **indispensable d'initialiser ces deux derniers champs** même s'ils ne sont pas utilisés, sous peine d'obtenir un armement au comportement “aléatoire”.

Armement standard

Nous appelons “**armement standard**” d'un signal sur un handler un armement correspondant au fait que

- le champ **sa_mask** est initialisé avec **tous des 0**.
- le champs **sa_flags** est égal à **0**.

Cela entraîne que

1. Pendant l'exécution du handler, aucun signal n'est masqué, mis à part le signal en cours de traitement pour lequel l'armement a été fait. N'importe quel autre signal interromperait le processus pendant l'exécution de ce handler.
2. Après le traitement de l'interruption,
 - (a) le processus **revient juste après l'appel système interrompu** (avec retour -1 de cet appel système et errno positionné à EINTR) ou **entre les 2 lignes de codes** (en mode utilisateur) où il a été interrompu.
 - (b) le **handler** mis en place **persiste** après traitement d'une première interruption.

5.3.2 Prototype du handler de signal

Le champ **sa_handler** est de type **void (*)(int)**, c'est-à-dire une fonction prenant en paramètre un int et ne retournant rien (void). Il peut prendre les valeurs suivantes :

- **SIG_DFL** si l'on souhaite ré-installer le comportement par défaut,
- **SIG_IGN** si l'on souhaite que le signal correspondant soit ignoré,
- le **nom d'une fonction** dont le prototype est obligatoirement

```
void    Nomfonction(int);
```

Deux remarques s'imposent :

1. Au moment de la réception d'un signal, la fonction **NomFonction** sera exécutée de manière **asynchrone** (on ne sait pas prédire quand) et recevra alors en **paramètre le numéro du signal** ayant provoqué l'interruption.
2. la fonction **NomFonction** n'est **appelée explicitement nulle part** dans le code de l'utilisateur. Il est donc impossible de tester une valeur de retour de cette fonction, d'où le type de retour **void**.

Exemple 5.3 (sigaction). Soit le programme suivant

```
***** TestSigaction *****/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void HandlerSIGINT(int sig);

int main()
{
    printf("PID = %d\n",getpid());

    struct sigaction A;
    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    int ret = pause();
    printf("ret = %d\n",ret);
    perror("Apres pause");

    exit(0);
}

void HandlerSIGINT(int sig)
{
    printf("\nJ'ai recu le signal no %d\n",sig);
}
```

dont un exemple d'exécution fournit

```
# ./TestSigaction
PID = 13597
^C
J'ai recu le signal no 2
ret = -1
Apres pause: Interrupted system call
# ./TestSigaction &
[1] 13605
PID = 13605
# kill -2 13605
J'ai recu le signal no 2
ret = -1
Apres pause: Interrupted system call
[1]+ Fini          ./TestSigaction
#
```

On constate que :

1. Après avoir armé SIGINT sur la fonction HandlerSIGINT, le processus s'endort sur l'appel système **pause**. Lors de la réception du signal SIGINT (son envoi est dû soit à un <CTRL-C> soit à l'utilisation de la commande **kill**), le processus exécute le handler, revient après l'appel système qui a retourné **-1** et **errno** a été positionné à **EINTR** (qui correspond au message "**Interrupted system call**").
2. Le comportement par défaut du processus vis-à-vis de SIGINT (fin du processus) a été remplacé par l'exécution de la fonction HandlerSIGINT.
3. Il n'est pas nécessaire de redéfinir la **structure sigaction**, celle-ci étant déjà définie dans le fichier **signal.h**.
4. Le **paramètre reçu** par la fonction HandlerSIGINT lors de son exécution est bien **2** qui correspond au signal SIGINT.
5. Les champs **sa_flags** et **sa_mask** de la structure A ont bel et bien été initialisés avant l'armement du signal. L'ensemble de signaux **sa_mask** étant vide, le seul signal masqué pendant l'exécution du handler est SIGINT. Si le processus avait reçu un autre signal pendant l'exécution de HandlerSIGINT, le signal aurait été interprété directement.

5.3.3 Masquage des signaux pendant l'exécution du handler

Le vecteur des signaux masqués est modifié automatiquement lors de l'exécution du handler qui a été mis en place par la fonction sigaction. En effet, l'**ensemble de signaux masqués pendant l'exécution d'un handler** est l'union

- du **signal en cours de traitement**,
- de l'ensemble des signaux présents dans **sa_mask**, et
- de l'**ensemble des signaux masqués avant la réception du signal**.

Par défaut, aucun signal n'est masqué.

Exemple 5.4 (sa_mask). Soit le programme suivant

```
***** TestSaMask.cpp *****
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void HandlerSIGINT(int sig);

int main()
{
    printf("PID = %d\n",getpid());

    struct sigaction A;
    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    // sigaddset(&A.sa_mask,SIGQUIT);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    while(1)
    {
        printf("Avant pause()\n");
        pause();
        printf("Apres pause()\n");
    }

    exit(0);
}

void HandlerSIGINT(int sig)
{
    printf("\nEntree HandlerSIGINT\n");
    sleep(10);
    printf("Sortie HandlerSIGINT\n");
}
```

dont un premier exemple d'exécution fournit

```
# ./TestSaMask
PID = 14599
Avant pause()
^C
Entree HandlerSIGINT
Sortie HandlerSIGINT
Apres pause()
Avant pause()
^C
Entree HandlerSIGINT
^C Sortie HandlerSIGINT

Entree HandlerSIGINT
Sortie HandlerSIGINT
Apres pause()
Avant pause()
^C
Entree HandlerSIGINT
^C\Quitter (core dumped)
#
```

On constate que :

1. Lors du 3ème <CTRL-C> (et donc de l'envoi du signal SIGINT au processus), le processus est en train d'exécuter la handler de signal HandlerSIGINT, il est endormi sur l'appel système **sleep**. Le signal **SIGINT** n'est alors pas interprété directement, il est **bloqué**. Le signal SIGINT est donc bien masqué pendant l'exécution du hander. Ce n'est que lorsque le processus sort du handler qu'il se rend compte que le signal SIGINT est pendant (reçu et non masqué) et qu'il interprète de nouveau le signal.
 2. Le <CTRL-\> provoque l'envoi du signal **SIGQUIT** au processus au moment où il est en train d'exécuter le handlerSIGINT. Le champ sa_mask étant vide, seul le signal SIGINT est masqué pendant l'exécution du handler. Le signal SIGQUIT est donc **interprété directement** et vu que le comportement par défaut du processus vis-à-vis de SIGQUIT est la fin du processus, celui-ci se termine immédiatement comme en témoigne le message "**Quitter (core dumped)**".

Un autre exemple d'exécution du même programme est le suivant :

```
# ./TestSaMask
PID = 14943
Avant pause()
^C           le nbre de ctrl c on s'en fou, un est suffisant ( on n'envoie qu'un signal )
Entree HandlerSIGINT
^C^C^C^CSortie HandlerSIGINT

Entree HandlerSIGINT
Sortie HandlerSIGINT
Apres pause()
Avant pause()
^QQuitter (core dumped)
#
```

On constate que

1. Lors du premier <CTRL-C> (envoi de SIGINT), le processus remet le vecteur des signaux reçus à 0 à la case correspondant à SIGINT et entre dans le HandlerSIGINT.
2. Lors du second <CTRL-C>, rien ne se passe visuellement car le signal SIGINT est masqué pendant l'exécution du handler. Cependant, ce **second signal SIGINT n'est pas perdu**, il est **bloqué** et apparaît dans le vecteur des signaux reçus.
3. Les 3ème, 4ème et 5ème <CTRL-C> provoquent l'envoi de 3 signaux SIGINT supplémentaires au processus. Cependant, ceux-ci sont **perdus** car le vecteur des signaux reçus est incapable de comptabiliser le nombre de signaux SIGINT reçus.
4. Lorsque le processus revient de sa première interruption, il se rend compte que le signal SIGINT est **pendant** (reçu et non masqué) et l'interprète directement en exécutant à nouveau le handler.

Un dernier exemple d'exécution du même programme, mais en décommentant la seule ligne commentée (**sigaddset...**), fournit :

```
# ./TestSaMask
PID = 15159
Avant pause()
^C
Entree HandlerSIGINT
^C~\^C~\Sortie HandlerSIGINT

Entree HandlerSIGINT
Sortie HandlerSIGINT
Quitter (core dumped)
#
```

On constate que

1. L'ajout du signal **SIGQUIT** à `sa_mask` fait que l'ensemble des **signaux masqués** pendant l'exécution de HandlerSIGINT est à présent constitué de SIGQUIT et de SIGINT.
2. Lors du premier <CTRL-C> (envoi de SIGINT), le processus remet le vecteur des signaux reçus à 0 à la case correspondant à SIGINT et entre dans le HandlerSIGINT.
3. Lors du second <CTRL-C>, rien ne se passe visuellement car le signal SIGINT est masqué pendant l'exécution du handler. Cependant, **ce signal SIGINT n'est pas perdu**, il est **bloqué** et apparaît dans le vecteur des signaux reçus.
4. Lors du premier <CTRL-\> (envoi de SIGQUIT), rien ne se passe visuellement car le signal SIGQUIT est masqué pendant l'exécution du handler. Cependant, **ce signal SIGQUIT n'est pas perdu**, il est **bloqué** et apparaît dans le vecteur des signaux reçus.
5. Le 3ème <CTRL-C> et le second <CTRL-\> n'ont aucun effet et les signaux **SIGINT et SIGQUIT sont perdus**.
6. Lorsque le processus revient de sa première interruption, il se rend compte que le signal **SIGINT est pendant** (reçu et non masqué) et l'interprète directement en exécutant à nouveau le handler. De retour de cette seconde interruption, le processus se rend compte que le signal **SIGQUIT est pendant** et l'interprète directement en exécutant le comportement par défaut provoquant la fin du processus.



5.3.4 Quelques variantes à l'armement standard

Le paramètre **sa_flags** permet de modifier le comportement du handler de signal **pendant** et **après** son exécution. Il peut, par exemple, prendre les valeurs

- **SA_RESETHAND** : dans ce cas, après l'exécution du handler mis en place, celui-ci est remplacé par le **comportement par défaut**. Il s'agit en quelque sorte d'un "one shot", le signal est armé sur un handler qui ne sera exécuté qu'une seule fois, après quoi le processus retrouvera son comportement par défaut vis-à-vis de ce signal.
- **SA_RESTART** : dans ce cas, l'**appel système interrompu** ne retourne pas -1 mais est **relancé** après l'exécution du handler. Après le traitement de l'interruption, le processus retourne donc sur l'appel système interrompu et non après.
- **SA_NODEFER** : dans ce cas, le **signal en cours** de traitement, et donc pour lequel l'armement a été fait, n'est **pas masqué** pendant l'exécution du handler. Le traitement d'une interruption associée à un signal particulier peut donc être interrompu par ce même signal.

Le champs **sa_flags** peut également être une **combinaison** de plusieurs de ces constantes en utilisant l'opérateur OU logique |.

Exemple 5.5 (SA_RESETHAND). Soit le programme suivant

```
***** TestSA_RESETHAND.cpp *****/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void HandlerSIGINT(int sig);

int main()
{
    printf("PID = %d\n",getpid());

    struct sigaction A;
    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = SA_RESETHAND;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    pause();
    pause();
    pause();
    pause();

    exit(0);
}

void HandlerSIGINT(int sig)
{
    printf("\nReception signal no %d\n",sig);
```

empiler
des
interrupt
ions
(ctrl c-)

}

dont un exemple d'exécution fournit

```
# ./TestSA_RESETHAND
PID = 16621
^C
Reception signal no 2
^C
#
```

On constate que :

1. Le premier <CTRL-C> (envoi de SIGINT donc) a interrompu le premier appel de la fonction **pause** et a provoqué l'**exécution du handler** mis en place. Le processus est alors revenu de son interruption et s'est endormi sur le second appel de pause.
2. Lors du second <CTRL-C>, le **comportement par défaut** ayant été remis en place, le signal SIGINT provoque l'interruption du second appel à la fonction **pause** et le comportement par défaut du processus vis-à-vis de SIGINT, c'est-à-dire la fin du processus.
3. Le processus ne revient donc jamais de la seconde interruption et le troisième appel de la fonction **pause** n'est jamais atteint.



Exemple 5.6 (SA_RESTART). Soit le programme suivant

```
***** TestSA_RESTART.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void HandlerSIGINT(int);

int main()
{
    struct sigaction A;
    int ret;
    char buffer[80];

    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction(1)");
        exit(1);
    }

    printf("Lecture clavier (1) :\n");
    ret = read(0,buffer,10);
    perror("Apres read");
    buffer[ret] = '\0';
    printf("Lu : %s (%d)\n",buffer,ret);

    A.sa_flags = SA_RESTART;
```

```

if (sigaction(SIGINT,&A,NULL) == -1)
{
    perror("Erreur de sigaction(2)");
    exit(1);
}

printf("Lecture clavier (2) :\n");
ret = read(0,buffer,10);
perror("Apres read");
buffer[ret] = '\0';
printf("Lu : %s (%d)\n",buffer,ret);

exit(0);
}

void HandlerSIGINT(int sig)
{
    printf("\nReception signal no %d\n",sig);
}

```

dont un exemple d'exécution fournit

```

# ./TestSA_RESTART
Lecture clavier (1) :
aaaa^C
Reception signal no 2
Apres read: Interrupted system call
Lu : (-1)
Lecture clavier (2) :
bbb^C
Reception signal no 2
rrrr^C
Reception signal no 2
xxx
Apres read: Interrupted system call
Lu : xxx
(4)
#

```

On constate que :

1. Lors du premier <CTRL-C> (envoi de SIGINT donc), le signal SIGINT a été **armé de manière standard**. L'appel système **read** a été interrompu, a retourné -1 et le processus a poursuivi son exécution après l'appel système.
2. Lors du second <CTRL-C>, le signal a été armé avec le flag **sa_flags=SA_RESTART**. Le second appel système **read** a été interrompu puis relancé, ce qui a permis d'encoder “rrrr” avant d'être interrompu à nouveau par le 3ème <CTRL-C>.
3. De nouveau, l'appel système a été relancé et l'utilisateur a pu finalement encoder “xxx\n”, d'où un nombre de caractères lus égal à **4** et le retour de ligne avant l'affichage de (4).

Exemple 5.7 (SA_NODEFER). Soit le programme suivant

```

***** TestSA_NODEFER.cpp *****
#include <stdio.h>
#include <stdlib.h>

```

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

void HandlerSIGINT(int);

time_t T1,T2;

int main()
{
    struct sigaction A;

    time(&T1);

    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = SA_NODEFER;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    while (1) pause();

    exit(0);
}

void HandlerSIGINT(int Sig)
{
    printf("\nEntree apres %ld secondes\n", time(&T2) - T1);
    sleep(5);
    printf("\nSortie apres %ld secondes\n", time(&T2) - T1);
}
```

dont un exemple d'exécution fournit

```
# ./TestSA_NODEFER
^C
Entree apres 4 secondes
^C
Entree apres 7 secondes

Sortie apres 12 secondes

Sortie apres 12 secondes
^C\Quitter (core dumped)
#
```

On constate que :

1. Lors du premier <CTRL-C> (envoi de SIGINT donc), le signal **SIGINT** a interrompu l'appel système **pause** du main.
2. Lors du second <CTRL-C>, le processus est en train d'exécuter le handler de SIGINT et est endormi sur l'appel système **sleep** qui est à son tour interrompu par le signal SIGINT. Ceci montre bien que le signal **SIGINT n'est plus masqué pendant l'exécution du handler** mis en place pour SIGINT.

3. Lors du retour de la seconde interruption (au bout de 12 secondes d'exécution), le processus se retrouve juste après le **sleep** de sa première interruption. Il sort alors enfin de sa première interruption et retourne sur l'appel système **pause** du main où il est interrompu par la réception de SIGQUIT.



5.3.5 Application : Gestion des processus zombis, le signal SIGCHLD

Une première application courante et importante de l'armement d'un signal est la **gestion asynchrone des processus fils zombis d'un processus**.

Lorsqu'un processus se termine, il envoie automatiquement un signal **SIGCHLD à son père**. Cependant, le comportement par défaut d'un processus vis-à-vis de SIGCHLD est de l'**ignorer**. Si le processus père continue son exécution ainsi, les processus fils terminés passent à l'état de **zombi** et **conservent une entrée dans la table des processus**. Le seul moyen de retirer les processus zombis de cette table est de faire exécuter l'appel système **wait** au processus père. Il suffit dès lors d'**armer**, dans le **processus père**, le signal **SIGCHLD** sur un handler exécutant la fonction **wait**.

Considérons un premier exemple où le processus père ne gère pas la fin de ses processus fils zombi.

Exemple 5.8 (Pas de gestion des fils zombi). Soit le programme suivant

```
***** TestZombi1.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main()
{
    pid_t idFils1,idFils2;

    if ((idFils1 = fork()) == -1) { perror("Erreur de fork"); exit(1); }

    if (idFils1 == 0)
    {
        // Fils 1
        printf("(FILS 1) pid=%d Je vis ma vie 8 secondes\n",getpid());
        sleep(8);
        exit(8);
    }

    if ((idFils2 = fork()) == -1) { perror("Erreur de fork"); exit(1); }

    if (idFils2 == 0)
    {
        // Fils 2
        printf("(FILS 2) pid=%d Je vis ma vie 5 secondes\n",getpid());
        sleep(5);
        exit(5);
    }

    // Pere
    printf("(PERE) Je realise une tache...\n");
    while(1) pause();

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./TestZombi1 &
[1] 12748
(PERE) Je realise une tache...
(FILS 1) pid=12750 Je vis ma vie 8 secondes
(FILS 2) pid=12751 Je vis ma vie 5 secondes
# ps
  PID TTY      TIME CMD
 6941 pts/2    00:00:00 bash
12748 pts/2    00:00:00 TestZombi1
12750 pts/2    00:00:00 TestZombi1
12751 pts/2    00:00:00 TestZombi1
12757 pts/2    00:00:00 ps
# ps
  PID TTY      TIME CMD
 6941 pts/2    00:00:00 bash
12748 pts/2    00:00:00 TestZombi1
12750 pts/2    00:00:00 TestZombi1 <defunct>
12751 pts/2    00:00:00 TestZombi1 <defunct>
12765 pts/2    00:00:00 ps
# kill -9 12748
[1]+ Processus arrêté ./TestZombi1
# ps
  PID TTY      TIME      CMD
 6941 pts/2    00:00:00 bash
12787 pts/2    00:00:00 ps
#
```

On constate que :

1. Le processus père a créé deux processus fils avant de s'endormir sur l'appel système **pause**.
2. Lors du premier **ps**, le processus père et les deux processus fils sont en cours d'exécution (le père est endormi sur le **pause** tandis que les deux fils sont endormis sur leur **sleep**).
3. Lors du second **ps**, les deux processus **fils** se sont **terminés** et ont envoyé le signal **SIGCHLD à leur père**, ce qui n'a eu **aucun effet** sur celui-ci (il l'ignore !). Dès lors, les deux fils sont passés à l'état de **zombi**, ce qui est indiqué par **<defunct>**.
4. L'exécution de la commande **kill** a pour effet de **tuer le processus père**. Les deux processus fils zombis deviennent **orphelins** et sont adoptés par le processus **init** qui les supprime directement de la table des processus. Ceci est confirmé par le troisième **ps**.



Il est tout de même hasardeux de laisser le processus **init (PID = 1)** s'occuper des fils zombis d'autres processus. En effet, le processus père pourrait ne jamais s'arrêter (comme un processus démon), la table des processus se remplirait indéfiniment de processus zombis jusqu'au moment où elle serait pleine. La création de nouveaux processus ne serait alors plus possible. Il est donc préférable que chaque processus gère proprement la fin de ses processus fils zombis en armant correctement le signal SIGCHLD.

Exemple 5.9 (Gestion correcte des fils zombis). Soit le programme suivant

```
***** TestZombi2.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

void HandlerSIGCHLD(int);

int main()
{
    pid_t idFils1,idFils2;
    struct sigaction A;

    // Armement de SIGCHLD
    A.sa_handler = HandlerSIGCHLD;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGCHLD,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    // Creation des fils
    if ((idFils1 = fork()) == -1) { perror("Erreur de fork"); exit(1); }

    if (idFils1 == 0)
    {
        // Fils 1
        printf("(FILS 1) pid=%d Je vis ma vie 8 secondes\n",getpid());
        sleep(8);
        exit(8);
    }

    if ((idFils2 = fork()) == -1) { perror("Erreur de fork"); exit(1); }

    if (idFils2 == 0)
    {
        // Fils 2
        printf("(FILS 2) pid=%d Je vis ma vie 5 secondes\n",getpid());
        sleep(5);
        exit(5);
    }

    // Pere
    printf("(PERE) Je realise une tache...\n");
    while(1) pause();

    exit(0);
}

void HandlerSIGCHLD(int Sig)
{
    int id, status;
    id = wait(&status);
    printf("\n(PERE) Suppression du fils %d de la table des processus\n",id);
```

exo4

```

if (WIFEXITED(status))
    printf("(PERE) Le fils %d s'est termine par un exit(%d)\n",id,WEXITSTATUS(status));
}

```

dont un exemple d'exécution fournit

```

# ./TestZombi2 &
[1] 13094
(PERE) Je realise une tache...
(FILS 1) pid=13096 Je vis ma vie 8 secondes
(FILS 2) pid=13097 Je vis ma vie 5 secondes
# ps
  PID TTY      TIME CMD
 6941 pts/2    00:00:00 bash
13094 pts/2    00:00:00 TestZombi2
13096 pts/2    00:00:00 TestZombi2
13097 pts/2    00:00:00 TestZombi2
13103 pts/2    00:00:00 ps
#
(PERE) Suppression du fils 13097 de la table des processus
(PERE) Le fils 13097 s'est termine par un exit(5)

(PERE) Suppression du fils 13096 de la table des processus
(PERE) Le fils 13096 s'est termine par un exit(8)

# ps
  PID TTY      TIME CMD
 6941 pts/2    00:00:00 bash
13094 pts/2    00:00:00 TestZombi2
13139 pts/2    00:00:00 ps
#

```

On constate que :

- Après avoir armé le signal SIGCHLD, le processus père a créé deux processus fils avant de s'endormir sur l'appel système **pause**.
- Lors du premier **ps**, le processus père et les deux processus fils sont en cours d'exécution (le père est endormi sur le **pause** tandis que les deux fils sont endormis sur leur **sleep**).
- Lorsque les processus **fils** se terminent, ils envoient le signal **SIGCHLD à leur père**. La fonction pause est alors interrompue et le processus père exécute le **handler de SIGCHLD** dans lequel il récupère toute l'information sur la fin du processus fils terminé et le supprime de la table des processus à l'aide de la fonction **wait** (qui n'est **pas bloquante** car le fils qui vient de se terminer est zombi). Ceci est confirmé par le second **ps**.
- La gestion des zombis par le processus père est dite **asynchrone** car il n'attend pas en permanence la fin d'un de ses fils en restant bloqué sur l'appel système **wait**. Il "vit sa vie" (ici en exécutant l'appel système **pause**) et il **ne sait pas à l'avance** quand il va recevoir le signal **SIGCHLD**, d'où l'asynchronisme.

5.4 Masquage de signaux : la fonction sigprocmask

Masquer un signal signifie que

- celui apparaît dans le **vecteur des signaux masqués** (appelé aussi **masque de signaux**) du processus,
- si celui-ci est reçu par le processus, il n'est **pas interprété** mais il n'est **pas perdu**, il est **bloqué** et apparaît alors dans le vecteur des signaux reçus. Si le signal est **démasqué**, il sera alors **interprété** par le processus.

Le champ **sa_mask** de la structure **sigaction** permet de modifier le vecteur des signaux masqués mais **uniquement pendant l'exécution du handler** de signal correspondant. Pour modifier le masque de signaux (ou le vecteur des signaux masqués donc) du processus en dehors des handlers de signaux, il est nécessaire d'utiliser la fonction :

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

où

- **oldset** est l'adresse d'un ensemble de signaux (de type **sigset_t**) permettant de récupérer en local l'**ancien vecteur des signaux masqués** du processus. Ce paramètre peut être **NULL**, dans ce cas, on ne récupère pas l'ancien masque de signaux du processus.
- **set** est l'adresse d'un ensemble de signaux (de type **sigset_t**) permettant de modifier le vecteur des signaux masqués du processus.
- **how** est un paramètre entier permettant de choisir comment utiliser le vecteur **set** pour modifier le vecteur des signaux masqués du processus :

block et unblock, complication, mieux vaux utiliser le premier

- **SIG_SETMASK** : l'ancien vecteur des signaux masqués du processus est tout simplement **remplacé par le vecteur set**.
- **SIG_BLOCK** : le vecteur des signaux masqués du processus correspond à l'union (OU logique bit à bit) entre l'ancien vecteur des signaux masqués et l'ensemble **set**. Cela correspond au fait que les signaux présents dans **set** sont **ajoutés** au vecteur des signaux masqués. Cela revient donc à **masquer les signaux présents dans set**.
- **SIG_UNBLOCK** : le vecteur des signaux masqués du processus correspond à l'ancien vecteur de signaux masqués dont on a **retiré** les signaux présents dans **set**. Cela revient donc à **démasquer les signaux présents dans set**.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,

- la valeur **-1** en cas d'erreur, et `errno` est positionné.

En cas d'erreur, les valeurs de `errno` les plus fréquentes sont

errno	signification
EINVAL	la valeur de <code>how</code> est invalide
EFAULT	<code>set</code> ou <code>oldset</code> pointent vers une adresse invalide
...	...

Remarquez qu'il est **impossible de masquer les signaux SIGKILL (9) et SIGSTOP (19)**. Cela permet de garder quoi qu'il arrive un accès au processus via une interruption. Au moment de l'appel de `sigprocmask`, ces signaux sont tout simplement retirés du masque de signal du processus sans provoquer d'erreur.

Exemple 5.10 (sigprocmask). Soit le programme suivant

```
***** TestSigprocmask.cpp *****/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

int main()
{
    printf("PID = %d\n",getpid());

    // Confection en local d'un masque de signal
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask,SIGINT);
    sigaddset(&mask,SIGQUIT);

    // Mise en place du nouveau masque de signal
    sigprocmask(SIG_SETMASK,&mask,NULL);

    pause();

    exit(0);
}
```

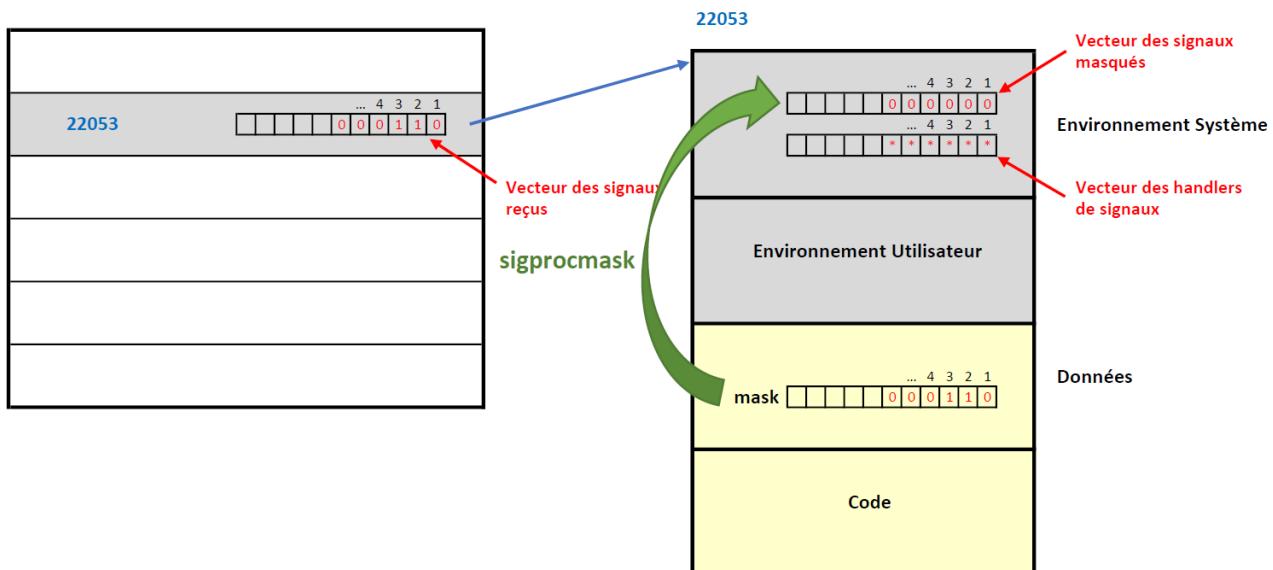
dont un exemple d'exécution fournit

```
# ./TestSigprocmask
PID = 22053
^C^C\^C^Z
[1]+ Stoppé ./TestSigprocmask
# kill -2 22053
# kill -3 22053
# ps
  PID TTY      TIME      CMD
  5181 pts/2    00:00:00 bash
22053 pts/2    00:00:00 TestSigprocmask
  22080 pts/2    00:00:00 ps
# kill -9 22053
[1]+ Processus arrêté ./TestSigprocmask
#
```

On constate que :

- Une variable locale **mask** est nécessaire afin de **construire en local** (dans la zone de données du processus) le masque de signal (dans cet exemple on veut masquer SIGINT et SIGQUIT) que l'on veut mettre en place. Une fois initialisée, la variable locale **mask** est transférée dans l'**environnement système** du processus, en remplacement du vecteur de signaux masqués du processus, à l'aide de la fonction **sigprocmask** avec le paramètre **how** égal à **SIG_SETMASK** :

Table des processus



- Le 3ème paramètre de **sigprocmask** étant **NULL**, on n'a pas voulu récupérer en local l'ancien masque de signaux du processus.
- Les <CTRL-C> (envoi de **SIGINT**) et <CTRL-\> (envoi de **SIGQUIT**) n'ont aucun effet visuel, mais ces deux signaux apparaissent dans le vecteur signaux reçus, ils sont donc **bloqués**. Le processus reste donc endormi sur l'appel système **pause**.
- Le <CTRL-Z> provoque l'envoi du signal **SIGSTOP** au processus. Vu que celui-ci n'est pas masqué, le processus l'interprète et le comportement par défaut est l'**arrêt (pas la fin !)** du processus. L'utilisateur récupère alors la main sur le terminal.
- L'envoi des signaux **SIGINT** et **SIGQUIT** par la commande **kill** n'a également aucun effet.
- Le processus se termine suite à la réception du signal **SIGKILL**.



5.4.1 Mise en place d'une section critique

Une **section critique** est une portion importante de code

- pendant laquelle le processus **ne peut pas être dérangé**, c'est-à-dire **ne peut pas être interrompu** par un signal,
- commençant par l'appel de la fonction **sigprocmask** réalisant le **masquage des signaux** par lesquels on ne veut pas être dérangé,

- se terminant par l'appel de la fonction **sigprocmask** remettant en place le masque de signaux existant avant la section critique.

Après la section critique, tous les signaux **pendants** (c'est-à-dire reçus et non masqués) sont **interprétés** les uns à la suite des autres avant que le processus puisse reprendre son exécution normale.

Exemple 5.11 (section critique). Soit le programme suivant

```
***** TestSectionCritique.cpp *****
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void Handler(int sig);

int main()
{
    printf("PID = %d\n",getpid());

    // Armement des signaux SIGINT et SIGQUIT
    struct sigaction A;
    A.sa_handler = Handler;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction(1)");
        exit(1);
    }

    if (sigaction(SIGQUIT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction(2)");
        exit(1);
    }

    // Confection en local d'un masque de signal
    sigset_t mask,oldMask;
    sigfillset(&mask);

    // Section critique
    sigprocmask(SIG_SETMASK,&mask,&oldMask);
    printf("Debut section critique...\n");
    sleep(10);
    printf("Fin section critique...\n");
    sigprocmask(SIG_SETMASK,&oldMask,NULL);

    printf("Apres section critique...\n");
    pause();

    exit(0);
}

void Handler(int sig)
{
    printf("\nReception du signal no %d\n",sig);
}
```

dont un exemple d'exécution fournit

```
# ./TestSectionCritique
PID = 23616
Debut section critique...
^C^CFin section critique...

Reception du signal no 3

Reception du signal no 2

Apres section critique...
^C
Reception du signal no 2
#
```

On constate que :

1. Les signaux **SIGINT** et **SIGQUIT** ont été armés sur le **même handler**, ce qui ne pose aucun problème.
2. Lors de l'appel du premier **sigprocmask**, la variable **oldMask** est utilisée pour **mémoriser le masque de signal avant la section critique**.
3. Les <CTRL-C> (envoi de **SIGINT**) et <CTRL-\> (envoi de **SIGQUIT**) n'ont aucun effet visuel pendant la **section critique**, mais ces deux signaux apparaissent dans le vecteur des signaux reçus, ils sont donc **bloqués**. Le processus reste donc endormi sur l'appel système **sleep** (considéré dans l'exemple comme la portion importante de code)
4. Pour sortir de la section critique, l'appel de la fonction **sigprocmask remet en place l'ancien masque** de signal **oldMask**.
5. Après la section critique, le processus scrute le vecteur des signaux reçus et se rend compte que les signaux **SIGINT** et **SIGQUIT** sont **pendants**. Ceux-ci sont donc **interprétés directement** et les handlers correspondants sont exécutés avant que le processus ne reprenne son exécution normale.
6. Le dernier <CTRL-C> provoque l'interruption de l'appel système **pause**, l'exécution du handler de **SIGINT** et enfin la fin du processus qui se termine par l'appel de **exit**.



Lors de la sortie d'une section critique, les signaux pendents ne sont pas traités dans l'**ordre chronologique** de leur arrivée et il n'y a aucune priorité entre les signaux. Le processus scrute simplement le vecteur des signaux reçus dans l'**ordre numérique** (inverse ici) des signaux et traitent les signaux pendents dans cet ordre.

5.4.2 Récupérer le vecteur des signaux pendents/reçus

Il est possible à tout moment à un processus de connaître l'**ensemble des signaux pendents**, c'est-à-dire l'ensemble des signaux qu'il a **reçus** et qu'il n'a **pas encore traités**, que ceux-ci soient masqués ou non. Pour cela il peut utiliser la fonction :

```
#include <signal.h>

int sigpending(sigset_t *set);
```

où `set` est l'adresse d'une variable locale au processus (zone de données) dans laquelle on souhaite récupérer l'ensemble des signaux reçus (et non encore traités).

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et `errno` est positionné.

En cas d'erreur, les valeurs de `errno` les plus fréquentes sont

errno	signification
EFAULT	set pointe vers une adresse invalide
...	...

Si le processus ne veut traiter qu'un seul des signaux reçus, il lui suffit alors de le **démasquer**. De plus, si le processus veut traiter les signaux reçus dans un ordre précis, il lui suffit à nouveau de démasquer ces signaux un par un dans l'ordre souhaité. Une fois démasqué, le signal reçu est immédiatement interprété.

Exemple 5.12 (sigpending). Soit le programme suivant

```
***** TestSigpending.cpp *****
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void afficheMasque(const sigset_t* masque,int val);

int main()
{
    printf("PID = %d\n",getpid());

    // Masquage de tous les signaux (sauf SIGSTOP et SIGKILL)
    sigset_t mask;
    sigfillset(&mask);
    sigprocmask(SIG_SETMASK,&mask,NULL);

    // Attente reception de signaux
    sleep(10);

    // Recuperation du vecteur des signaux recus
    sigset_t pending;
    if (sigpending(&pending) == -1)
    {
        perror("Erreur de sigpending");
        exit(1);
    }

    printf("\nSignaux recus : ");
    afficheMasque(&pending,20);

    exit(0);
}
```

```

}

void afficheMasque(const sigset_t* masque,int val)
{
    for (int sig=val ; sig>0 ; sig--) printf("%d",sigismember(masque,sig));
    printf("\n");
}

```

dont un exemple d'exécution fournit

```

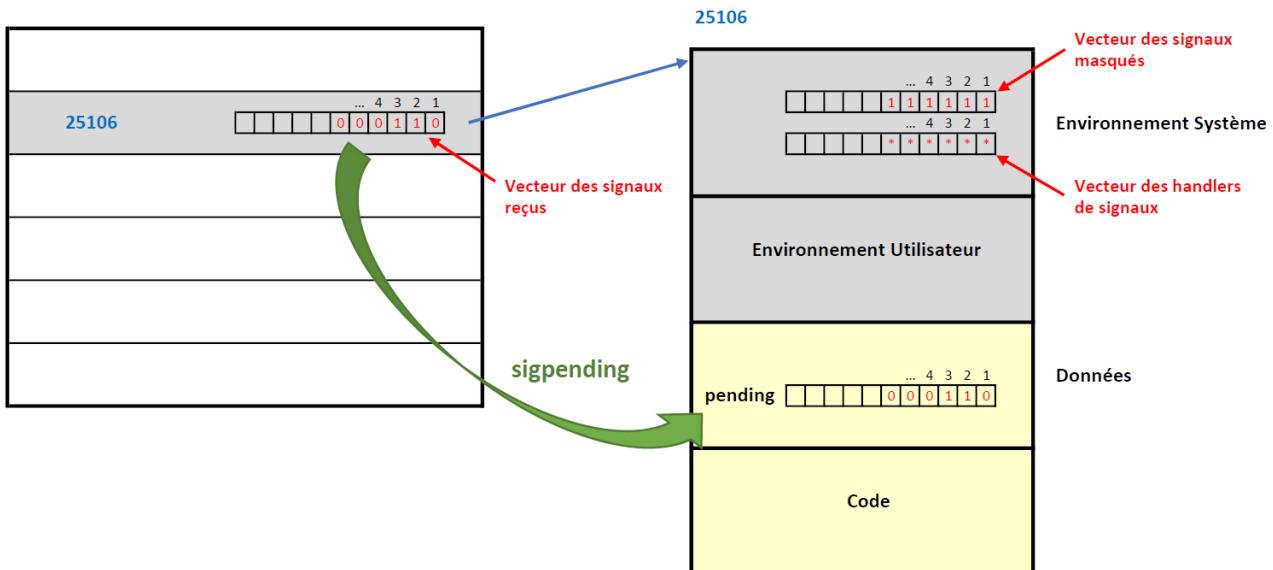
# ./TestSigpending
PID = 25106
^C^C^\
Signaux recus : 0000000000000000110
# ./TestSigpending &
[1] 25281
PID = 25281
# kill -2 25281
# kill -5 25281
Signaux recus : 000000000000000010010
[1]+ Fini ./TestSigpending
#

```

On constate que :

1. on retrouve la fonction `afficheMasque` de l'exemple 5.2 permettant d'afficher le contenu d'un ensemble de signaux.
2. Tous les signaux (sauf SIGQUIT et SIGKILL) sont **masqués** au début afin de pouvoir envoyer au processus quelques signaux sans qu'il ne les interprète directement, sans quoi il aurait été impossible de visualiser un vecteur de signaux reçus non vide.
3. L'envoi des signaux, soit via des <CTRL-C> ou <CTRL-\>, soit par la commande `kill`, n'a aucun effet visuel, mais les signaux reçus apparaissent bien dans le vecteur des signaux reçus, ils sont donc **bloqués**. Le processus reste donc endormi sur l'appel système `sleep`.
4. La fonction **`sigpending`** récupère bien l'ensemble des signaux reçus dans sa variable locale **`pending`** comme l'illustre le schéma suivant :

Table des processus



5.5 Les sauts non locaux

Lors de la réception de certains signaux, il est parfois nécessaire qu'un processus puisse reprendre son exécution normalement. Plus précisément, après le traitement d'une interruption, le processus doit pouvoir reprendre son exécution à un **endroit bien précis du code**, avec un **masque de signaux précis**. On dit encore que le processus reprend dans un **contexte d'exécution précis** après le traitement de l'interruption.

Pour réaliser ce genre de chose, on peut utiliser les fonctions suivantes :

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

où

- **env** est une variable qui mémorise un **contexte d'exécution** (un endroit précis dans le code, ainsi que le masque de signaux en cours à cet endroit).
- **savesigs** est un paramètre permettant de sauvegarder le **masque de signaux en cours** :
 - s'il est **différent de 0**, la fonction sigsetjmp **mémorise le masque de signaux** en cours dans la variable **env**.
 - s'il est **égal à 0**, le masque de signal en cours n'est **pas mémorisé** dans **env**.
- **val** est une valeur entière arbitraire (à spécifier par le programmeur donc) qui sera transmise à la fonction sigsetjmp (via sa **valeur de retour**) pour l'informer, par exemple, d'où le

saut a été réalisé.

La fonction **sigsetjmp** mémorise donc le **contexte actuel** (c'est-à-dire l'endroit où la fonction sigsetjmp se trouve, que l'on pourrait appeler "point de retour du saut non local", et éventuellement le masque de signaux actuel) et retourne la valeur 0. Le processus peut alors continuer son exécution normalement, traiter des interruptions, etc... Dès qu'il rencontre la fonction **siglongjmp**, le processus "saute" jusque l'endroit où la fonction **sigsetjmp** a été exécutée. Tout se passe alors comme si la fonction sigsetjmp était exécutée une nouvelle fois mais à présent son **code de retour** est **val**. Le contexte sauvegardé dans la variable env est alors restauré (avec ou sans le masque de signal, selon que savesigs est différent de 0 ou non).

La fonction **siglongjmp** n'a **pas de retour**, elle ne revient jamais car le processus "saute" sur le sigsetjmp correspondant.

Exemple 5.13 (siglongjmp). Soit le programme suivant

```
***** TestSiglongjmp.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
    sigjmp_buf contexte;
    int ret,ch;

    // Mise en place d'un point de retour du saut
    if ((ret = sigsetjmp(contexte,1)) != 0)
    {
        printf("\nRetour du saut %d...\n",ret);
    }

    printf("Retour 10 (1 ou 0) ? ");
    fflush(stdin); scanf("%d",&ch);
    if (ch == 1) siglongjmp(contexte,10);

    printf("Retour 20 (1 ou 0) ? ");
    fflush(stdin); scanf("%d",&ch);
    if (ch == 1) siglongjmp(contexte,20);

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./TestSiglongjmp
Retour 10 (1 ou 0) ? 1

Retour du saut 10...
Retour 10 (1 ou 0) ? 0
Retour 20 (1 ou 0) ? 1

Retour du saut 20...
Retour 10 (1 ou 0) ? 0
Retour 20 (1 ou 0) ? 0
#
```

On constate que :

1. **Un seul contexte** a été mémorisé par la fonction **`sigsetjmp`** dans la variable **contexte** et **deux sauts non locaux** sont possibles (deux appels différents à **`siglongjmp`**).
2. Les valeurs **10** et **20** sont totalement **arbitraires** et auraient pu être choisies autrement.
3. Selon le saut réalisé, le **retour** de la fonction **`sigsetjmp`** vaut 10 ou 20, lui permettant de savoir d'où le saut a été réalisé.



Exemple 5.14 (`sigsetjmp`). Soit le programme suivant

```
***** TestSigsetjmp.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
    sigjmp_buf contexte1, contexte2;
    int ret,ch;

    // Mise en place d'un premier point de retour de saut
    printf("Sauvegarde contexte1\n");
    if ((ret = sigsetjmp(contexte1,1)) != 0)
    {
        printf("\nRetour au contexte 1 (ret = %d)... \n",ret);
    }

    // Mise en place d'un second point de retour de saut
    printf("Sauvegarde contexte2\n");
    if ((ret = sigsetjmp(contexte2,1)) != 0)
    {
        printf("\nRetour au contexte 2 (ret = %d)... \n",ret);
    }

    printf("Retour au contexte1 (1), au contexte2 (2) ou pas de retour (0) ? ");
    fflush(stdin); scanf("%d",&ch);
    if (ch == 1) siglongjmp(contexte1,10);
    if (ch == 2) siglongjmp(contexte2,10);

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./TestSigsetjmp
Sauvegarde contexte1
Sauvegarde contexte2
Retour au contexte1 (1), au contexte2 (2) ou pas de retour (0) ? 2

Retour au contexte 2 (ret = 10)...
Retour au contexte1 (1), au contexte2 (2) ou pas de retour (0) ? 1

Retour au contexte 1 (ret = 10)...
Sauvegarde contexte2
Retour au contexte1 (1), au contexte2 (2) ou pas de retour (0) ? 0
#
```

On constate que :

1. **Deux contextes** ont été mémorisés par la fonction **`sigsetjmp`** dans les variables **contexte1** et **contexte2**.
2. La valeur **10** est à nouveau totalement **arbitraire** et aurait pu être choisie autrement.
3. Selon le contexte choisi, le **`siglongjmp`** “sauté” sur le bon point de retour sauvegardé.

■

L'inconvénient des sauts non locaux est qu'un usage trop fréquent **diminue la lisibilité** des programmes. Il est conseillé de les réservier toujours au même type de circonstances dans une application donnée.

5.5.1 Exemple : Traitement correct du signal SIGFPE

L'exemple qui suit montre un **traitement correct du signal SIGFPE** que le noyau envoie à un processus qui réalise une opération mathématique erronée comme une **division par zéro**. L'utilisation d'un saut non local va permettre au processus de “sauter” à un endroit du programme où l'erreur sera gérée correctement.

Exemple 5.15 (gestion du signal SIGFPE par un saut non local). Soit le programme suivant qui calcule en boucle le résultatat de la division de 2 nombres entiers encodés par l'utilisateur :

```
***** TestGestionSIGFPE.cpp *****/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf contexte;

void HandlerSIGFPE(int s);

int main()
{
    int D,d,q;
    int ret;

    // Armement du signal SIGFPE
    struct sigaction A;
    A.sa_handler = HandlerSIGFPE;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGFPE,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    // Mise en place d'un point de retour du saut
    if ((ret = sigsetjmp(contexte,1)) != 0)
    {
        printf("Erreur mathematique !!! (ret = %d)\n",ret);
    }

    // Boucle principale du programme
    while(1)
    {
        printf("Dividende = ");
        fflush(stdin); scanf("%d",&D);

        printf("Diviseur = ");
        fflush(stdin); scanf("%d",&d);

        q = D/d;

        printf("Quotient = %d\n\n",q);
    }

    exit(0);
}

void HandlerSIGFPE(int s)
{
    siglongjmp(contexte,5);
}
```

dont un exemple d'exécution fournit

```
# ./TestGestionSIGFPE
Dividende = 12
Diviseur = 4
Quotient = 3

Dividende = 7
Diviseur = 0
Erreur mathematique !!! (ret = 5)
Dividende = ^C
#
```

On constate que :

1. Si le signal SIGFPE n'avait pas été armé sur le HandlerSIGFPE, la ligne de code `q=D/d;` avec `d` égal à `0` aurait provoqué l'envoi du signal **SIGFPE** par le noyau et le **comportement par défaut** de ce signal est la **fin du processus**, ce que l'on ne souhaite pas ici.
2. Lorsque le signal SIGFPE est reçu et traité par le processus, le **siglongjmp** fait "sauter" le processus en un endroit du code qui gère correctement l'erreur de division par 0 (par l'affichage d'un message ici) que l'on peut qualifier de "gestionnaire d'erreur".
3. la variable **contexte** est déclarée en **globale** car elle doit être accessible dans le main mais également dans le handler de SIGFPE.
4. Le paramètre **5** du **siglongjmp** est à nouveau arbitraire et aurait pu être choisi autrement.



5.5.2 Mémorisation du masque de signaux dans le contexte

Nous allons à présent observer l'effet du paramètre **savesigs** sur le comportement du saut non local.

Exemple 5.16 (savesigs). Soit le programme suivant

```
***** TestSavesigs.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>

void HandlerSIGINT(int);

sigjmp_buf contexte;

int main()
{
    char buffer[80];
    int ret;

    // Armement du signal SIGINT
    struct sigaction A;
    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
```

```

{
    perror("Erreur de sigaction");
    exit(1);
}

// Mise en place d'un point de retour du saut
ret = sigsetjmp(contexte,1);
printf("ret : %d\n",ret);
printf("Entrer une premiere chaine : ");
fgets(buffer,80,stdin);
printf("Lu : --%s--\n",buffer);

// Mise a jour du point de retour de saut
ret = sigsetjmp(contexte,0);    il faut pas mettre de 0, car sinon, à l'inverse du 1, ça conserve le masque du
printf("ret : %d\n",ret);        singint, donc même en spammant de ctr c on peut pas quitter le code
printf("Entrer une seconde chaine : ");
fgets(buffer,80,stdin);
printf("Lu : --%s--\n",buffer);

exit(0);
}

void HandlerSIGINT(int sig)
{
    printf("\nReception de SIGINT\n");
    siglongjmp(contexte,sig);
}

```

dont un exemple d'exécution fournit

```

# ./TestSavesigs
ret : 0
Entrer une premiere chaine : aaa^C
Reception de SIGINT
ret : 2
Entrer une premiere chaine : aaabbb^C
Reception de SIGINT
ret : 2
Entrer une premiere chaine : aaabbccc
Lu : --aaabbccc
--
ret : 0
Entrer une seconde chaine : xxx^C
Reception de SIGINT
ret : 2
Entrer une seconde chaine : xxxyyy^C^C^C^C^C^Z
[2]+ Stoppé ./TestSavesigs
#

```

On constate que :

1. La même variable **contexte** a servi à mémoriser **deux contextes différents** du processus. Cependant, le second appel à **sigsetjmp** écrase l'ancien contenu de la variable **contexte**. Une fois le second point de retour mis en place, il n'est alors plus possible de retourner au premier point de retour.
2. Lors du **premier appel à sigsetjmp**, le paramètre **savesigs** est à 1 (**différent de 0** donc), tout se passe correctement. L'interruption par <CTRL-C> (réception de SIGINT) est

gérée, puis le processus retourne à son premier point de retour sans aucun souci, et cela autant de fois que la première saisie est interrompue par <CTRL-C>.

3. Lors du **second appel à sigsetjmp**, le paramètre **savesigs** est à **0**, et là, par contre, après le traitement d'une première interruption, il est **impossible d'interrompre une nouvelle fois la seconde saisie clavier**. Vu que le masque de signaux n'a pas été mémorisé dans la variable contexte, le saut du handler de signal vers le second point de retour s'est fait en **conservant le masque de signaux appliqué pendant l'exécution du handler**. Or, pendant l'exécution du handler de SIGINT, le signal **SIGINT est masqué**! Lors de la dernière saisie clavier, le signal SIGINT est donc masqué.



Afin d'éviter des soucis de masque de signaux non restitué correctement, il est donc préférable de toujours mettre le paramètre **savesigs** à une valeur différente de 0.

5.6 Les groupes de processus

Comme il existe des groupes d'utilisateurs, il existe également la notion de **groupe de processus**. Celle-ci est particulièrement pratique lorsque l'on désire envoyer un signal, ou interrompre plusieurs processus simultanément.

Chaque processus est membre d'un groupe de processus. Un groupe de processus est identifié par un **numéro** appelé **"Process Group ID"** ou **PGID**. Si le PID d'un processus est égal au PGID d'un groupe, on dit que ce processus en est le **leader**. De plus,

- un processus lancé en **ligne de commande** crée **son propre groupe** dont il est le leader et dont le PGID correspond à son propre PID,
- un processus **fils** appartient par défaut au **groupe de processus de son père**.

Il est possible à un processus de connaître son PGID à l'aide de la fonction :

```
#include <unistd.h>
pid_t getpgrp(void);
```

dont la valeur de retour est de même type que le PID d'un processus.

Exemple 5.17 (getpgrp). Soit le programme suivant

```
***** TestGetpgrp.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    pid_t idFils;

    if ((idFils = fork()) == -1)
    {
        perror("Erreur de fork()");
        exit(1);
    }
```

```

}

if (idFils != 0)
{
    // Pere
    printf("(PERE) pid : %d groupe : %d\n",getpid(),getpgrp());
    exit(0);
}

// Fils
printf("(FILS) pid : %d groupe : %d\n",getpid(),getpgrp());
exit(0);
}

```

dont un exemple d'exécution fournit

```
# ./TestGetpgrp
(PERE) pid : 32394 groupe : 32394
(FILS) pid : 32395 groupe : 32394
#
```

On constate que :

1. Le processus père de PID 32394 a créé un nouveau groupe de **PGID** égal à 32394.
2. Le processus fils, créé par l'appel système fork, se trouve dans le **même groupe de processus que son père**.



5.6.1 Attacher un processus à un groupe

Un processus peut changer de groupe à l'aide de la fonction :

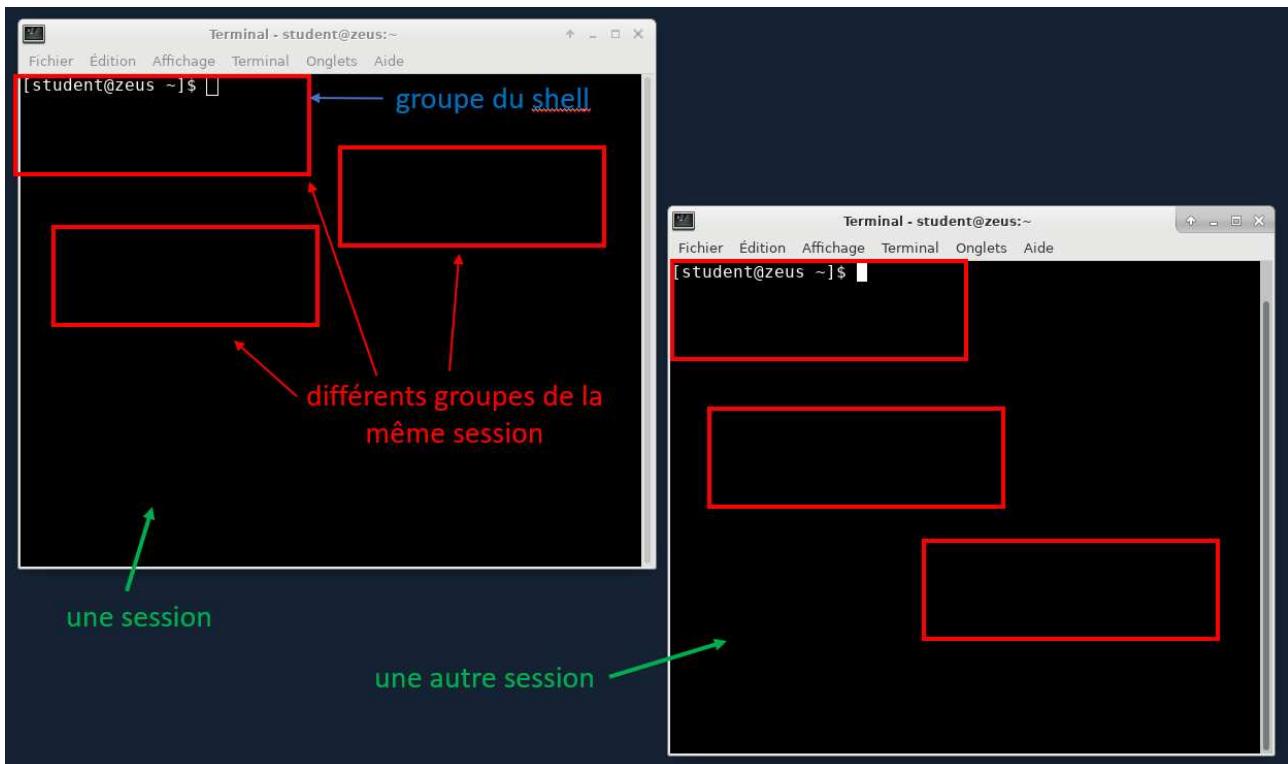
```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

où

- **pid** est le **PID** du processus que l'on souhaite changer de groupe.
- **pgid** est le **PGID** du groupe dans lequel on désire placer le processus pid.

Pour pouvoir transférer un processus d'un groupe à un autre groupe, ces deux groupes doivent faire partie de la **même session**. La notion de session est illustrée sur le schéma suivant :



Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EPERM	On a essayé de déplacer un processus dans un groupe d'une session différente, ou de changer le groupe d'un fils du processus appelant qui se trouve dans une autre session, ou de modifier le groupe d'un leader de session.
EINVAL	pgid est inférieur à 0
...	...

Une application pratique de cette fonction est de faire appartenir un ou plusieurs processus au **groupe du shell**. Ainsi, un signal envoyé par un **caractère spécial** sur le terminal (**<CTRL-C>** pour **SIGINT** ou **<CTRL-\>** pour **SIGQUIT**) ou même par la fermeture du terminal par un clic sur la croix (envoi du signal **SIGHUP**) est reçu par l'ensemble des processus faisant partie du **groupe de processus du shell**.

Exemple 5.18 (setpgid). Soit le programme suivant

```
***** TestSetpgid.cpp *****
#include <stdio.h>
#include <unistd.h>
```

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

void HandlerSIGINT(int);

int main()
{
    // Armement de SIGINT
    struct sigaction A;
    A.sa_handler = HandlerSIGINT;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGINT,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    int idFils;

    // Attachement du processus au groupe du pere
    if (setpgid(getpid(),getppid()) == -1)
    {
        perror("Erreur de setpgid");
        exit(1);
    }

    for (int i=1 ; i<4 ; i++)
    {
        if ((idFils = fork()) == -1)
        {
            perror("Erreur de fork");
            exit(1);
        }

        if (idFils == 0)
        {
            // Fils i
            printf("(FILS %d) pid:%d groupe:%d Traitement en arriere plan...\n",i,getpid(),getpgrp());
            pause();
        }
    }

    // Pere
    exit(0);
}

void HandlerSIGINT(int Sig)
{
    printf("\n (PID = %d) Reception du signal SIGINT\n",getpid());
    exit(1);
}
```

dont un exemple d'exécution fournit

```
# ./TestSetpgid
(FILS 2) pid:2205 groupe:792 Traitement en arriere plan...
(FILS 3) pid:2206 groupe:792 Traitement en arriere plan...
(FILS 1) pid:2204 groupe:792 Traitement en arriere plan...
# ps
 PID TTY      TIME CMD
 792 pts/2    00:00:00 bash
2204 pts/2    00:00:00 TestSetpgid
2205 pts/2    00:00:00 TestSetpgid
2206 pts/2    00:00:00 TestSetpgid
2213 pts/2    00:00:00 ps
#
 (PID = 2206) Reception du signal SIGINT

 (PID = 2205) Reception du signal SIGINT

 (PID = 2204) Reception du signal SIGINT
^C
# ps
 PID TTY      TIME CMD
 792 pts/2    00:00:00 bash
2233 pts/2    00:00:00 ps
#
```

On constate que :

1. Le processus a créé 3 fils avant de se terminer. Les trois fils réalisent un traitement en arrière plan qui correspond simplement ici à l'attente sur la fonction **pause**. Pour ces 3 fils, le signal **SIGINT a été armé**. En fait, c'est le père qui a armé ce signal mais les 3 processus fils ont hérité de cet armement.
2. Le processus s'est **rattaché au groupe de son propre père**, c'est-à-dire le **shell** de PID égal à **792**, leader de son propre groupe. Les 3 fils font dès lors partie du même groupe de processus.
3. Le **<CTRL-C>** dans le terminal a eu pour effet d'envoyer le signal **SIGINT à tous les membres du groupe du processus shell**. En effet, les 3 processus fils ont exécuté le handler de SIGINT. Remarquez qu'un seul <CTRL-C> et donc **un seul signal SIGINT** suffit pour que les 3 fils reçoivent le signal.
4. Si la fonction **setpgid** n'avait pas été exécutée, le <CTRL-C> n'aurait eu aucun effet sur les 3 processus fils qui seraient restés endormis sur l'appel système pause.



Nous illustrons à présent le problème du rattachement d'un processus à un groupe qui ne fait pas partie de la même session que lui.

Exemple 5.19 (erreur de setpgid). Soit le programme suivant

```
***** Prog1.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

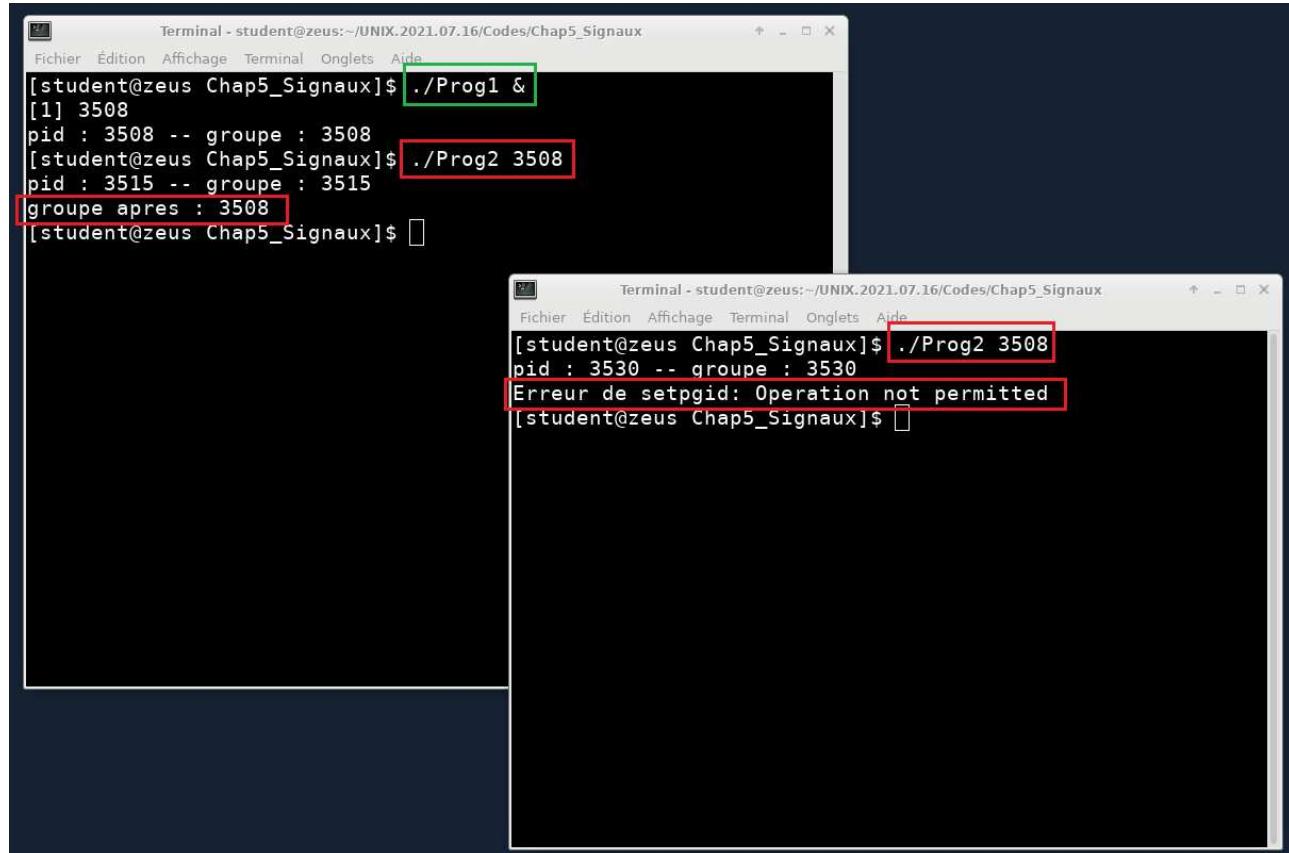
int main()
{
```

```
printf("pid : %d -- groupe : %d\n",getpid(),getpgrp());  
pause();  
  
exit(0);  
}
```

et le programme

```
***** Prog2.cpp *****/  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int main(int argc,char* argv[])
{
    printf("pid : %d -- groupe : %d\n",getpid(),getpgrp());  
  
    if (setpgid(getpid(),atoi(argv[1])) == -1)
    {
        perror("Erreur de setpgid");
        exit(1);
    }
  
  
    printf("groupe apres : %d\n",getpgrp());  
  
    exit(0);
}
```

dont des exemples d'exécution fournissent



```
Terminal - student@zeus:~/UNIX.2021.07.16/Codes/Chap5_Signaux  
Fichier Édition Affichage Terminal Onglets Aide  
[student@zeus Chap5_Signaux]$ ./Prog1 &  
[1] 3508  
pid : 3508 -- groupe : 3508  
[student@zeus Chap5_Signaux]$ ./Prog2 3508  
pid : 3515 -- groupe : 3515  
groupe apres : 3508  
[student@zeus Chap5_Signaux]$ 
```



```
Terminal - student@zeus:~/UNIX.2021.07.16/Codes/Chap5_Signaux  
Fichier Édition Affichage Terminal Onglets Aide  
[student@zeus Chap5_Signaux]$ ./Prog2 3508  
pid : 3530 -- groupe : 3530  
Erreur de setpgid: Operation not permitted  
[student@zeus Chap5_Signaux]$ 
```

On constate que :

1. Prog2 tente de se rattacher au groupe de processus dont le PGID est reçu en premier argument, en particulier au groupe de processus de Prog1 ici (**PGID = 3508**).
2. Lorsque le processus Prog2 est lancé dans la **même session** que Prog1, le rattachement de Prog2 au groupe de Prog1 se fait **sans problème**.
3. Lorsque le processus Prog2 est lancé dans une **autre session** que Prog1, le rattachement de Prog2 au groupe de Prog1 **échoue**.



5.6.2 Envoi d'un signal à un groupe de processus

Un processus peut envoyer un signal à un processus ou directement à tous les membres d'un groupe de processus. Pour cela, il doit utiliser la fonction :

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

int kill(pid_t pid, int sig);
```

où

- **pid** est le pid du processus à qui l'on veut envoyer un signal,
- **sig** est le numéro de signal que l'on souhaite envoyer.

Notez que cette fonction est particulièrement **mal nommée** car elle **ne tue pas le processus cible**. Si celui-ci meurt à la suite de l'appel de kill, c'est à cause de son comportement vis-à-vis du signal reçu.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès (au moins un signal a été envoyé),
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EINVAL	Le numéro de signal est invalide.
EPERM	Le processus appelant n'a pas l'autorisation d'envoyer un signal aux processus cibles.
ESRCH	Le processus ou le groupe de processus n'existe pas.
...	...

Remarquez qu'un processus n'a le droit d'envoyer un signal qu'à un autre processus ayant le même propriétaire.

En réalité, la fonction **kill** permet encore d'autres actions en fonction des valeurs de ses paramètres :

pid	sig	action
> 0	> 0	Le signal sig est envoyé au processus pid .
= 0	> 0	Le signal sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.
> 0	= 0	Aucun signal n'est envoyé mais les conditions d'erreur sont vérifiées ; ceci peut être utilisé pour vérifier l'existence du processus pid .
= -1	> 0	Le signal sig est envoyé à tous les processus sauf celui de PID 1 (init). Seuls les processus ayant le même propriétaire que le processus appelant reçoivent le signal.
< -1	> 0	Le signal sig est envoyé à tous les processus du groupe dont l'identifiant est -pid .
< -1	= 0	Aucun signal n'est envoyé mais les conditions d'erreur sont vérifiées ; ceci peut être utilisé pour vérifier l'existence du groupe de processus d'identifiant -pid .

Le paramètre **pid** fixé à **-1** peut être utilisé de façon très utile. Cela permet à un utilisateur de terminer tous les processus dont il est propriétaire. Cela est tout autant possible avec la **commande kill** qui possède des paramètres similaires. Par exemple :

```
# kill -9 -1
```

envoie le signal numéro 9 (SIGKILL) à tous les processus de l'utilisateur qui exécute la commande. Tous ses processus sont immédiatement terminés et il sort de session.

Exemple 5.20 (kill sur un processus). Soit le programme suivant, qui illustre l'envoi d'un signal par un processus à son fils :

```
***** TestKillProcessus.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void HandlerSIGINT(int);

int main()
{
    pid_t idFils;
    struct sigaction A;

    if ((idFils = fork()) == -1)
    {
        perror("Erreur de fork");
        exit(1);
    }

    if (idFils == 0)
    {
        // Fils
        printf("(FILS) pid=%d Armement de SIGINT\n",getpid());
        A.sa_handler = HandlerSIGINT;
        sigemptyset(&A.sa_mask);
```

```

A.sa_flags = 0;

if (sigaction(SIGINT,&A,NULL) == -1)
{
    perror("Erreur de sigaction");
    exit(1);
}

printf("(FILS) pid=%d Attente d'un signal\n",getpid());
pause();

exit(0);
}

// Pere
sleep(2);
printf("(PERE) Emission d'un SIGINT au processus fils %d\n",idFils);

if (kill(idFils,SIGINT) == -1)
{
    perror ("Erreur de kill");
    exit(1);
}

exit(0);
}

void HandlerSIGINT(int Sig)
{
    printf("\n(FILS) pid=%d Reception du signal SIGINT\n",getpid());
}

```

dont un exemple d'exécution fournit

```

# ./TestKillProcessus
(FILS) pid=7173 Armement du SIGINT
(FILS) pid=7173 Attente d'un signal
(PERE) Emission d'un SIGINT au processus fils 7173

(FILS) pid=7173 Reception du signal SIGINT
#

```

On constate que :

1. Le processus fils a armé le signal SIGINT avant de s'endormir sur l'appel système **pause**.
2. La fonction **kill** a ici été utilisée pour envoyer un signal à un seul processus, le fils du processus appelant.



Exemple 5.21 (kill sur un groupe de processus). Soit le programme suivant, qui illustre l'envoi d'un signal par un processus à un groupe de processus :

```

***** TestKillGroupe.cpp *****
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

```

```
#include <wait.h>
#include <signal.h>

int main()
{
    pid_t idFils1,idFils2,idFils3;
    pid_t id;
    pid_t groupe;

    if ((idFils1 = fork()) == -1) { perror("Erreur de fork(1)"); exit(1); }

    if (idFils1 == 0)
    {
        // Fils 1
        setpgid(getpid(),getpid());

        printf("(FILS 1) pid=%d Attente d'un signal\n",getpid());
        pause();
        exit(0);
    }

    groupe = idFils1;

    if ((idFils2 = fork()) == -1) { perror("Erreur de fork(2)"); exit(1); }

    if (idFils2 == 0)
    {
        setpgid(getpid(),groupe);

        printf("(FILS 2) pid=%d Attente d'un signal\n",getpid());
        pause();
        exit(0);
    }

    if ((idFils3 = fork()) == -1) { perror("Erreur de fork(3)"); exit(1); }

    if (idFils3 == 0)
    {
        setpgid(getpid(),groupe);

        printf("(FILS 3) pid=%d Attente d'un signal\n",getpid());
        pause();
        exit(0);
    }

    // Pere
    printf("(PERE) Attend la fin d'un des fils...\n");
    id = wait(NULL);
    printf("(PERE) Fils termine : %d\n",id);
    printf("(PERE) Envoi de SIGINT au groupe de processus\n");
    kill(-groupe,SIGINT);

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./TestKillGroupe &
[1] 7869
(PERE) Attend la fin d'un des fils...
(FILS 1) pid=7871 Attente d'un signal
(FILS 2) pid=7872 Attente d'un signal
(FILS 3) pid=7873 Attente d'un signal
# ps
  PID TTY      TIME      CMD
 6941 pts/2    00:00:00 bash
 7869 pts/2    00:00:00 TestKillGroupe
7871 pts/2    00:00:00 TestKillGroupe
7872 pts/2    00:00:00 TestKillGroupe
7873 pts/2    00:00:00 TestKillGroupe
 7879 pts/2    00:00:00 ps
# kill -9 7872
(PERE) Fils termine : 7872
(PERE) Envoi de SIGINT au groupe de processus
[1]+ Fini ./TestKillGroupe
# ps
  PID TTY      TIME      CMD
 6941 pts/2    00:00:00 bash
 7892 pts/2    00:00:00 ps
#
```

On constate que :

1. Le processus a créé **3 processus fils** pour ensuite s'endormir sur l'appel système **wait**, dans l'attente de la fin d'un de ses fils.
2. Le **premier fils a créé un groupe de processus** (identifiant **groupe**) dont il est devenu le **leader**, tandis que les deux autres fils se rattachent au groupe du premier fils.
3. L'exécution de la commande **kill** a eu pour effet d'envoyer le signal **SIGKILL** à un des 3 processus fils, celui de PID **7872**. Ceci a eu pour effet de terminer ce processus et de débloquer le processus père qui a alors envoyé le **signal SIGINT au groupe** de processus de ses fils.
4. Le comportement par défaut des processus fils vis-à-vis de SIGINT étant la fin du processus, tous les processus fils se sont alors terminés. Ceci est bien confirmé par l'exécution du second **ps**.



5.7 Quelques fonctions utiles

5.7.1 La fonction pause

La fonction, déjà abordée plusieurs fois ci-dessus,

```
#include <unistd.h>

int pause();
```

endort indéfiniment un processus jusqu'à la **réception d'un signal** non masqué. Après traitement du signal, le processus reprend son exécution juste après l'appel de pause, à condition bien sûr que l'exécution du handler n'ait pas provoqué la fin du processus. Le retour de la fonction pause est **-1** et **errno** est positionné à **EINTR**.

5.7.2 La fonction sleep

La fonction, déjà abordée plusieurs fois ci-dessus,

```
#include <unistd.h>

unsigned int sleep(unsigned int nb_sec);
```

endort le processus appelant pendant **nb_sec** secondes ou jusqu'à la **réception d'un signal** non masqué. Deux comportements sont alors possibles :

1. la fonction **n'est pas interrompue** par un signal. Dans ce cas, le retour de la fonction est **0** au terme de l'attente de **nb_sec** secondes.
2. la fonction est **interrompue** par un signal. Dans ce cas, **errno** est positionné à **EINTR** et le retour est égal au **nombre de secondes restantes** lors de la réception du signal. Cela permettrait, par exemple, au processus interrompu de relancer la fonction sleep pour compléter l'attente qui a été interrompue.

Exemple 5.22 (sleep). Soit le programme suivant :

```
***** TestSleep.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void HandlerSIGINT(int);

int main()
{
    pid_t idFils;
    struct sigaction A;
    int ret;

    if ((idFils = fork()) == -1)
    {
        perror("Erreur de fork");
        exit(1);
    }

    if (idFils == 0)
    {
        // Fils
        printf("(FILS) pid=%d Armement de SIGINT\n",getpid());
        A.sa_handler = HandlerSIGINT;
        sigemptyset(&A.sa_mask);
        A.sa_flags = 0 ;

        if (sigaction(SIGINT,&A,NULL) == -1)
```

```

{
    perror("Erreur de sigaction");
    exit(1);
}

printf("(FILS) pid=%d Attente de 5 secondes\n",getpid());
if ((ret = sleep(5)) != 0)
{
    perror("sleep interrompu");
    printf("Temps restant : %d\n",ret);
}

exit(0);
}

// Pere
printf("(PERE) Attente de 2 secondes\n");
ret = sleep(2);
printf("(PERE) ret = %d\n",ret);
printf("(PERE) Emission d'un SIGINT au processus fils %d\n",idFils);

if (kill(idFils,SIGINT) == -1)
{
    perror ("Erreur de kill");
    exit(1);
}

exit(0);
}

void HandlerSIGINT(int Sig)
{
    printf("\n(FILS) pid=%d Reception du signal SIGINT\n",getpid());
}

```

dont un exemple d'exécution fournit

```

# ./TestSleep
(PERE) Attente de 2 secondes
(FILS) pid=8986 Armement de SIGINT
(FILS) pid=8986 Attente de 5 secondes
(PERE) ret = 0
(PERE) Emission d'un SIGINT au processus fils 8986

(FILS) pid=8986 Reception du signal SIGINT
sleep interrompu: Interrupted system call
Temps restant : 3
#

```

On constate que :

1. Le processus a créé un processus fils qui a armé **SIGINT**.
2. Après une **attente non interrompue de 2 secondes** (**ret = 0**), le père a envoyé le signal **SIGINT à son fils** qui a donc **interrompu son attente de 5 secondes**.
3. Après traitement de l'interruption, le processus fils est revenu juste après l'appel système **sleep** qui a retourné la valeur **ret = 3 = 5 - 2**, c'est-à-dire le **temps restant** lors de la réception du signal SIGINT.



5.7.3 Gestion d'un "Time Out" : la fonction `alarm`, le signal SIGALRM

Il est possible pour un processus de demander au noyau une **temporisation** avant d'être prévenu à l'aide d'une **interruption**. En d'autres mots, un processus peut demander au système d'exploitation de lui envoyer le signal **SIGALRM** après un certain délai de temps choisi, à charge alors au processus d'**armer correctement** le signal SIGALRM sous peine de se terminer prématurément (comportement par défaut d'un processus vis-à-vis de SIGALRM). Entre la demande de temporisation et la réception du signal, le processus peut alors continuer son exécution **sans être bloqué**.

Pour ce faire, un processus peut utiliser la fonction :

```
#include <unistd.h>
unsigned int alarm(unsigned int nb_sec);
```

question examen : non bloquant

où **nb_sec** est le **nombre de secondes demandé avant la réception du signal SIGALRM** (qui sera donc envoyé automatiquement par le système au processus appelant). Cette fonction n'est **pas bloquante** et le processus continue son exécution directement après son appel.

Selon le cas, deux possibilités existent lors de l'appel de `alarm` (avec **nb_sec > 0**) :

1. Si **aucune temporisation** n'a été demandée auparavant, la fonction `alarm` retourne **0** et le signal **SIGALRM** sera reçu par le processus **nb_sec** secondes plus tard.
2. Si une demande de **temporisation est en cours**, la fonction `alarm` **annule cette temporisation**, retourne le **nombre de secondes restantes** avant la temporisation annulée, et, à nouveau, le signal **SIGALRM** sera reçu par le processus **nb_sec** secondes plus tard.

Si le paramètre **nb_sec** est égal à **0**, la **temporisation en cours** est **annulée** et le retour correspond à nouveau à un des deux cas ci-dessus.

Exemple 5.23 (alarm). Soit le programme suivant :

```
***** TestAlarm.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void HandlerSIGALRM(int);

int main()
{
    char buffer[80];
    struct sigaction A;
    int ret;

    A.sa_handler = HandlerSIGALRM;
    sigemptyset(&A.sa_mask);
    A.sa_flags = 0;

    if (sigaction(SIGALRM,&A,NULL) == -1)
    {
        perror("Erreur de sigaction");
        exit(1);
    }

    // Demande d'une temporisation de 20 secondes
    alarm(20);
    printf("Entrer une chaine (max. 20 secondes) : ");
    fgets(buffer,80,stdin);

    ret = alarm(0);
    printf("Vous avez repondu dans les temps, il restait %d secondes.\n",ret);
    printf("Lu : %s\n",buffer);

    exit(0);
}

void HandlerSIGALRM(int Sig)
{
    printf("\nTime out... Trop tard...\n");
    exit(1);
}
```

dont deux exemples d'exécution fournissent

```
# ./TestAlarm
Entrer une chaine (max. 20 secondes) : hello
Vous avez repondu dans les temps, il restait 15 secondes.
Lu : hello

# ./TestAlarm
Entrer une chaine (max. 20 secondes) :
Time out... Trop tard...
#
```

On constate que :

1. Lors de la **1ère exécution**, la chaîne de caractères "hello" a été encodée au bout de 5 secondes. Il restait donc **15 secondes** avant que le signal SIGALRM ne soit envoyé au

processus. L'appel à **alarm(0)** a annulé la temporisation en cours et a donc retourné **15**.

2. Lors de la **2ème exécution**, aucune chaîne de caractères n'a été encodée. Au bout des 20 secondes, le signal **SIGALRM** a interrompu la fonction **fgets** et le processus a exécuté le handler de SIGALRM dans lequel il s'est terminé par un **exit**.



Chapitre 6

Communication entre processus - Les IPC

6.1 Introduction

6.1.1 Généralités

Les **IPC** ("Inter Process Communication") sont des mécanismes qui permettent à plusieurs processus indépendants de communiquer entre eux ou de synchroniser leurs exécutions. Ils sont au nombre de 3 :

1. les **files de messages** : permettent à des processus distincts d'**envoyer/recevoir des données** de manière (a)synchrone. Ces données doivent être vues comme des paquets de bytes appelés "messages" dans lesquels le programmeur met ce qu'il veut. Contrairement à un fichier qui est lu comme un flot continu de données non structuré, les messages sont structurés et sont toujours envoyés/lus entièrement. De manière imagée, on peut encore dire que les files de messages sont un mécanisme de communication par **boîtes aux lettres**. Les messages ne sont pas nécessairement émis à destination d'un processus particulier, mais dans une file de messages dans laquelle le correspondant viendra lire des données quand il le désirera.
2. les **mémoires partagées** : il s'agit d'une **zone de mémoire** allouée (un peu comme un "malloc" le ferait) mais accessible par plusieurs processus différents, chaque processus possédant son propre pointeur vers la zone de mémoire commune. Chaque processus partageant cette mémoire peut y lire ou y écrire sans appel système particulier, juste comme un accès mémoire ordinaire. En revanche, un certain nombre de précautions devront être prises pour organiser les accès concurrents. Pour cela, on utilise
3. les **sémaphores** : permettent à plusieurs processus de **synchroniser leur exécution**. Dans leur forme la plus simple, ils peuvent être vus comme des "drapeaux" qu'un processus lève ou descend afin de prévenir les autres processus qu'il utilise une ressource commune. On peut encore les voir comme un système de "**feux rouges**" régulant les accès concurrents à une ressource commune.

Ces 3 mécanismes sont indépendants mais ils ont une série de propriétés communes.

6.1.2 Eléments communs à tous les IPC

Un IPC, que ce soit une file de messages, une mémoire partagée ou un ensemble sémaphores, possède

- une **table système** (à l'image de la table des processus ou de la table des fichiers ouverts) qui présente une entrée distincte pour chaque file de message, mémoire partagée

pour aider (pas de fichier)
corrompus etc.)

ou sémaphore. Chaque entrée possède toutes les caractéristiques de l'IPC. Il y a donc une table des files de messages, une table des mémoires partagées et une table des sémaphores.

- une **clé** qui est simplement un **entier partagé** entre les processus qui veulent communiquer via cet IPC. Cette clé peut être partagée via un fichier commun, ou tout autre moyen. Une analogie peut être faite avec le port d'une communication réseau mise en place entre plusieurs processus.
- une **structure commune**, décrite ci-dessous, et qui reprend les caractéristiques communes à tous les IPC.

Les caractéristiques communes à tous les IPC sont reprises dans la **structure ipc_perm** située dans le fichier /usr/include/sys/ipc.h :

```
/* Data structure used to pass permission information to IPC operations. */
struct ipc_perm
{
    key_t __key;                      /* Key. */
    uid_t uid;                         /* Owner's user ID. */
    gid_t gid;                         /* Owner's group ID. */
    uid_t cuid;                        /* Creator's user ID. */
    gid_t cgid;                        /* Creator's group ID. */
    unsigned short int mode;           /* Read/write permission. */
    ...
};
```

où

- **__key** est la clé de l'IPC, de type **key_t**,
- **uid** est le propriétaire de l'IPC,
- **gid** est le groupe contenant le propriétaire de l'IPC,
- **cuid** est le créateur de l'IPC,
- **cgid** est le groupe contenant le créateur de l'IPC,
- **mode** est le mode d'accès à l'IPC, en lecture et/ou écriture, à l'image du mode d'ouverture d'un fichier.

6.2 Contrôle des IPC en ligne de commande

Le contrôle des IPC est très important. En effet, les IPC sont des éléments

- présents sur le système **“entre les processus”** (ils ne font pas partie de l'espace mémoire d'un processus en particulier),
- utilisés par plusieurs processus,

- pouvant continuer à exister même si les processus qui les utilisaient n'existent plus,
- en nombres limités sur le système.

Chaque utilisateur se doit de vérifier régulièrement l'état de ses IPC et de les supprimer manuellement si nécessaire, afin de libérer ces ressources qui redeviennent disponibles pour d'autres utilisateurs.

6.2.1 Visualiser les IPC : la commande ipcs

La commande **ipcs** permet de visualiser les caractéristiques des IPC. Elle présente de nombreuses options :

Option	Signification
-a	affiche les caractéristiques de tous les IPC. Il s'agit aussi de l'option par défaut
-q	affiche les caractéristiques de toutes les files de messages uniquement
-m	affiche les caractéristiques de toutes les mémoires partagées uniquement
-s	affiche les caractéristiques de tous les sémaphores uniquement
-l	affiche les limites des IPC sur le système, comme par exemple le nombre maximal de files de messages autorisées simultanément
-u	affiche un résumé des informations sur les IPC
-i id	affiche toutes les caractéristiques de l'IPC dont l'identifiant est id . Il s'agit donc d'une option avec paramètre.
...	...

Plusieurs de ces options peuvent être utilisées simultanément.

Exemple 6.1 (ipcs). Une première exécution de la commande ipcs fournit :

```
# ipcs
----- Files de messages -----
clef      msqid      propriétaire perms    octets utilisés messages
...
----- Segment de mémoire partagée -----
clef      shmid      propriétaire perms    octets nattch états
...
----- Tableaux de sémaphores -----
clef      semid      propriétaire perms    nsems
...
#
```

On constate que :

1. Utilisée avec ou sans l'option **-a**, la commande **ipcs** affiche des informations sur tous les IPC.

2. On voit apparaître quelques **caractéristiques communes** à tous les IPC (**clé, propriétaire et permissions**) présentes dans la structure **ipc_perm** de chaque IPC. Les autres caractéristiques sont propres à chaque IPC et seront abordées plus loin.

Un autre exemple d'exécution est

```
# ipcs -q

----- Files de messages -----
clef      msqid      propriétaire perms    octets utilisés messages
...
#
```

On constate que l'option **-q** a permis de n'afficher que les files de messages.

Autre exemple avec l'option **-l** :

```
# ipcs -l

----- Limites de messages -----
nombre maximal de files système = 32000
taille maximale de message (octet) = 8192
taille maximale par défaut de file (octet) = 16384

----- Limites de la mémoire partagée -----
nombre maximal de segments = 4096
taille maximale de segment (kilooctet) = 18014398509465599
total de mémoire partagée maximal (kilooctet) = 18014398442373116
taille minimale de segment (octet) = 1

----- Limites des sémaphores -----
nombre maximal de tableaux = 32000
nombre maximal de sémaphores par tableau = 32000
nombre maximal de sémaphores système = 1024000000
nombre maximal d'opérations par appel semop = 500
valeur maximal de sémaphore = 32767

#
```

L'option **-l** fournit les **limites des IPC autorisées** sur le système. Par exemple, le nombre maximal de files de messages autorisées simultanément sur le système est égal à 32000 et la taille maximale d'un message est de 8192 octets.

Et finalement, voici un dernier exemple avec l'option **-u** :

```
# ipcs -u

----- États des messages -----
files allouées = 0
en-têtes utilisées = 0
espace utilisé = 0 octets

----- État de la mémoire partagée -----
segments alloués = 92
pages alloués = 48226
pages résidentes = 17916
pages d'échange = 0
performance de l'espace d'échange = 0 tentatives      0 succès

----- États des sémaphores -----
tableaux utilisés = 0
sémaphores alloués = 0

#
```

On observe dans cet exemple qu'il n'y a pas de file de messages, ni de sémaphore utilisé. Par contre, 92 mémoires partagées ont été allouées sur le système.



6.2.2 Supprimer les IPC : la commande ipcrm

La commande **ipcrm** permet de **supprimer un ou plusieurs IPC** simultanément. Voici sa forme générale :

```
ipcrm [ -M key | -m id | -Q key | -q id | -S key | -s id ] ...
```

où

– **-M** ou **-m** correspond à une **mémoire partagée** :

- **-M** doit être suivi de la clé **key** de l'IPC
- **-m** doit être suivi de l'identifiant **id** de l'IPC

– **-Q** ou **-q** correspond à une **file de messages** :

- **-Q** doit être suivi de la clé **key** de l'IPC
- **-q** doit être suivi de l'identifiant **id** de l'IPC

– **-S** ou **-s** correspond à un **sémaphore** :

- **-S** doit être suivi de la clé **key** de l'IPC
- **-s** doit être suivi de l'identifiant **id** de l'IPC

Nous reviendrons plus loin sur la notion d'identifiant non encore abordée jusqu'ici. Il s'agit d'options avec paramètre et on peut en utiliser plusieurs sur la même ligne de commande.

Exemple 6.2 (ipcrm). Voici quelques exemples d'exécution de la commande ipcrm :

```
# ipcs -q -s

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 0      student     600    0          0
0x00000929 32769 student     600    0          0
0x00000d80 65538 student     600    0          0

----- Tableaux de sémaphores -----
clef      semid  propriétaire perms nsems
0x00000929 65536 student     600    3
0x0000092d 98305 student     600    3

# ipcrm -Q 0x4d2
# ipcs -q -s

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x00000929 32769 student     600    0          0
0x00000d80 65538 student     600    0          0

----- Tableaux de sémaphores -----
clef      semid  propriétaire perms nsems
0x00000929 65536 student     600    3
0x0000092d 98305 student     600    3

# ipcrm -q 32769 -S 0x92d
# ipcs -q -s

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x00000d80 65538 student     600    0          0

----- Tableaux de sémaphores -----
clef      semid  propriétaire perms nsems
0x00000929 65536 student     600    3

#
```



En regard de l'exemple précédent, nous pouvons faire deux remarques importantes :

1. Deux **IPC de même type** ne peuvent **pas** avoir la **même clé**.
2. Deux **IPC de type différent** peuvent avoir la **même clé**.

6.3 Les files de messages

Comme déjà mentionné dans l'introduction,

- les files de messages permettent à des processus distincts/indépendants d'**envoyer/recevoir des données** de manière **(a)synchrone**. En particulier, un processus peut s'endormir afin d'attendre l'arrivée d'un message ou alors aller lire un message au moment où il le désire.
- Ces données doivent être vues comme des "**paquets de bytes**" appelés **messages** dans lesquels le programmeur met ce qu'il veut. Contrairement à un fichier qui est lu comme un flot continu de données non structurées, les **messages** sont **structurés** et sont **toujours envoyés/lus entièrement**.
- De manière imagée, on peut encore dire que les files de messages sont un mécanisme de communication par **boîtes aux lettres**. Les messages ne sont pas nécessairement émis à destination d'un processus particulier, mais dans une file de messages dans laquelle le correspondant viendra lire des données quand il le désirera.

Une **table système** est dédiée à la gestion des files de messages.

6.3.1 Structure interne d'une file de messages

La table système des files de messages comprend une entrée distincte pour chaque file de messages et chaque entrée est structurée par la structure suivante :

```
struct msqid_ds
{
    struct ipc_perm msg_perm; /* structure describing operation permission */

    struct msg *msg_first;    /* first message on queue */
    struct msg *msg_last;     /* last message in queue */

    time_t msg_stime;         /* time of last msgsnd command */
    time_t msg_rttime;        /* time of last msgrcv command */
    time_t msg_ctime;         /* time of last change */

    msglen_t __msg_cbytes;    /* current number of bytes on queue */
    msgqnum_t msg_qnum;       /* number of messages currently on queue */
    msglen_t msg_qbytes;      /* max number of bytes allowed on queue */

    pid_t msg_lspid;          /* pid of last msgsnd() */
    pid_t msg_lrpid;          /* pid of last msgrcv() */

    ...
};
```

où

- **msg_perm** est la structure commune à tous les IPC, reprenant le créateur, le propriétaire, les droits, ...

- **msg_stime** et **msg_rtime** sont respectivement les dates/heures des derniers envoi et lecture d'un message sur la file de messages.
- **msg_ctime** est la date/heure de création et/ou dernière modification de paramètres de la file de messages.
- **msg_qnum** est le **nombre de messages présents actuellement** sur la file de messages.
- **_msg_cbytes** est la somme des tailles (en bytes) de tous les messages présents actuellement sur la file de messages.
- **msg_qbytes** est le nombre de bytes (somme des tailles de tous les messages) que la file de messages peut contenir.
- **msg_lspid** et **msg_lrpid** sont respectivement les PID des derniers processus qui ont envoyé et lu un message sur la file de messages.
- **msg_first** et **msg_last** sont respectivement les pointeurs de tête et de queue de la liste chaînée dynamique qui constitue la **structure interne** (au niveau du **noyau**) de la file de messages.

La file de messages est donc représentée en mémoire par une **liste chaînée dynamique** de structures (de type **struct msg**, dites “**opaque**”) et qui est **non accessible explicitement** par le programmeur. L'envoi et la réception de messages sur la file de messages modifiera cette liste chaînée mais se fera via des **appels systèmes** décrits plus loin.

Chaque **noeud** de cette **liste chaînée dynamique** est représenté par la **structure msg** définie par

```
struct msg
{
    struct msg *msg_next; /* next message on queue */
    long msg_type;
    short msg_ts; /* message text size */
    char *msg_spot; /* message text address */
};
```

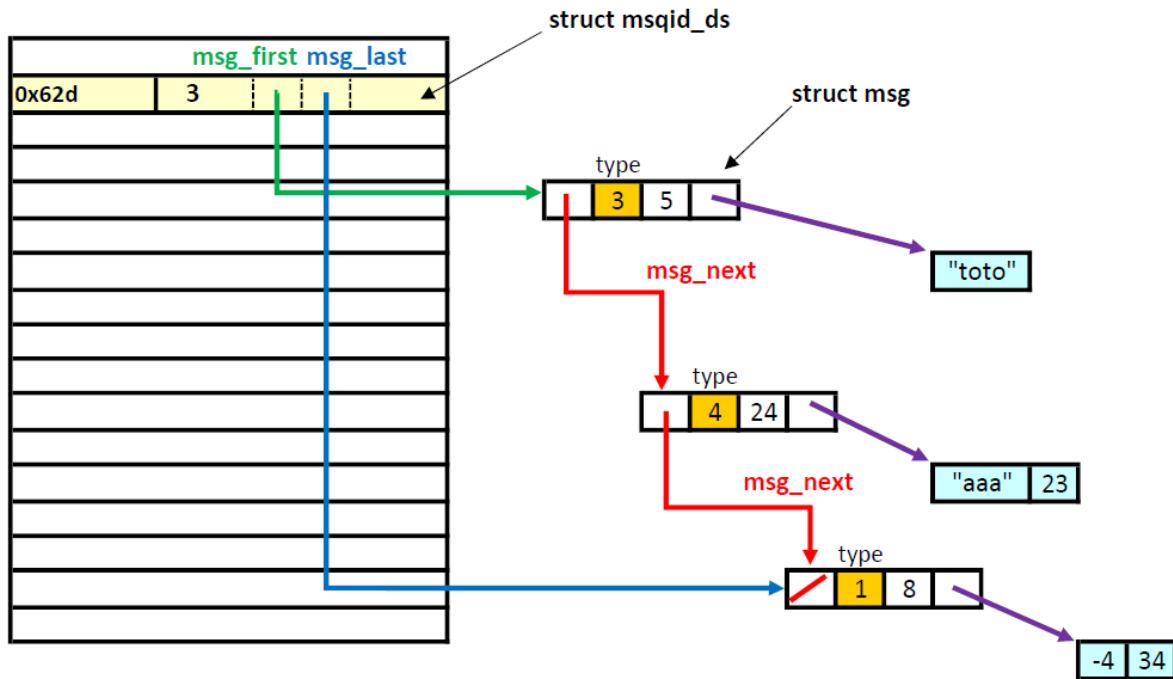
où

- **msg_next** est le pointeur vers le noeud suivant de la liste chaînée.
- **msg_type** est appelé le “**type de message**”. Il s'agit d'un entier positif important sur lequel nous revenons plus loin.
- **msg_spot** est un **pointeur** vers la **charge utile** du message ; en d'autres mots, vers le **paquet de bytes** d'un message dont nous avons déjà parlé plus haut.
- **msg_ts** est la **taille** du **paquet de bytes** constituant la **charge utile** du message.

Le schéma ci-dessous illustre la structure interne d'une file de messages, de clé 0x62d, contenant 3 messages :

TYPE = IMPORTANCE MSG

Table des files de messages

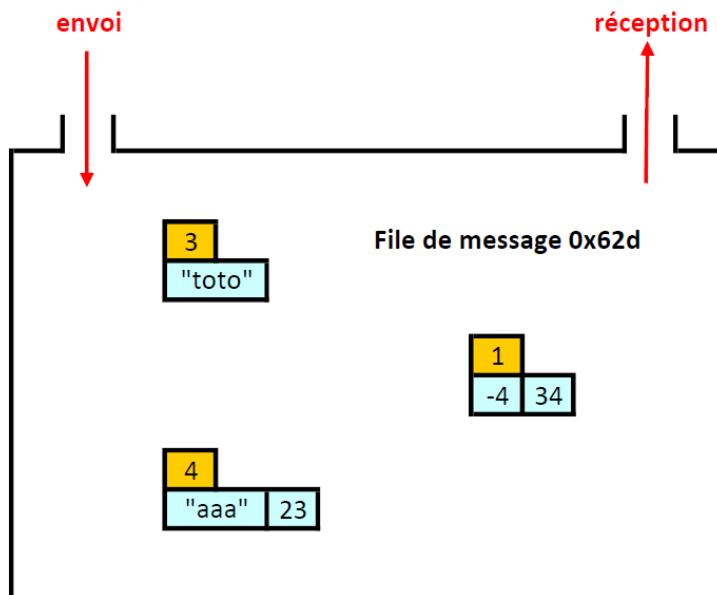


Il est important de remarquer que la **structure msg** n'est **pas un message**! Il s'agit de la structure qui fait office de **noeud** de la liste chaînée, chaque noeud pointant vers le message proprement dit (la "**charge utile**" en bleu clair) qui n'a pas de structure pré définie. Les 3 messages présents sur la file de messages ont les caractéristiques suivantes :

1. **Message de type 3** dont la charge utile de **5 octets** est
 - (a) une chaîne de caractères (**char[5]**) contenant "toto" (5 octets utilisés sur les 5 faisant partie de la charge utile).
2. **Message de type 4** dont la charge utile de **24 octets** est
 - (a) une chaîne de caractères (**char[20]**) contenant "aaa" (4 octets utilisés sur les 20 faisant partie de la charge utile),
 - (b) un entier (**int**) de 4 octets contenant la valeur 23.
3. **Message de type 1** dont la charge utile de **8 octets** est
 - (a) un entier (**int**) de 4 octets contenant la valeur -4,
 - (b) un entier (**int**) de 4 octets contenant la valeur 34.

6.3.2 Une espèce de "boîte aux lettres"

Du point de vue du programmeur, une file de messages peut être vue comme une simple **boîte aux lettres** et, d'une manière imagée, la file de messages de l'exemple précédent peut être représentée par



Chaque message peut ainsi être vu comme une **enveloppe**

1. dont le **contenu** est la **charge utile** (en bleu clair sur le schéma) du message,
2. dont l'**étiquette** collée dessus est le **type du message** (en orange sur le schéma).

6.3.3 Type d'un message

Le type d'un message est un **entier positif (non nul)** dont la signification est **arbitraire** et peut être **choisie par le programmeur**. C'est l' "étiquette que l'on colle sur l'enveloppe" du message et elle n'est pas prise en compte dans le calcul de la **taille du message**. La taille d'un message correspond à la charge utile, c'est-à-dire au contenu de l'enveloppe.

Voici quelques exemples d'utilisation possible du type du message :

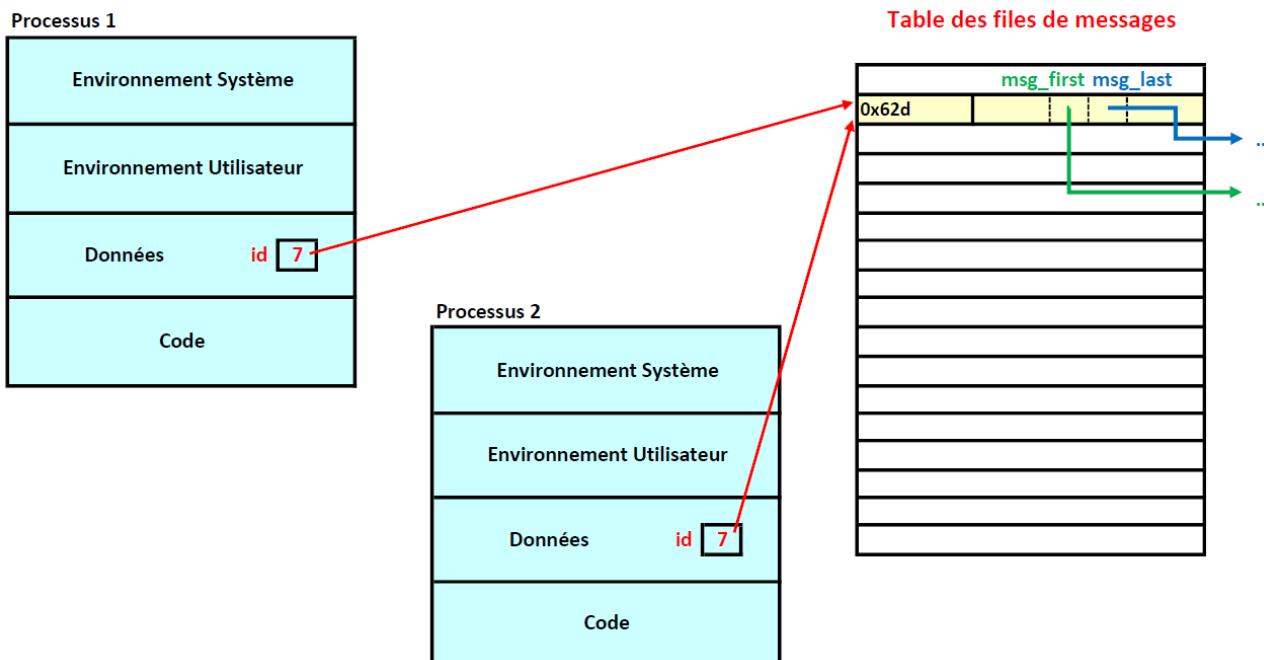
- Le type (bien que devant être initialisé lors de l'envoi du message) n'est **pas utilisé**. Les messages sont envoyés et lus sans en tenir compte. Ils sont simplement lus dans l'ordre d'arrivée et la file de messages se comporte alors comme une simple liste FIFO (First In First Out).
- Le type correspond à un **ordre de priorité**. Par exemple, plus le type est petit, plus le message a de l'importance et doit être traité en priorité. Un processus peut alors choisir de lire les messages de priorité haute avant ceux de priorité plus faible.
- Le type correspond au **PID du processus destinataire**. Ainsi, il est possible de "router" un message vers un destinataire précis. Un processus, connaissant son propre PID, peut alors choisir de ne lire que les messages dont le type correspondent à son propre PID. Il s'agit d'un des usages les plus fréquents.
- ...

6.3.4 Identifiant d'une file de messages

Au niveau programmation, une file de messages ne se manipule pas directement en utilisant sa clé. On peut faire l'analogie avec la lecture/écriture dans un fichier. Celui-ci ne se manipule pas directement avec le nom du fichier. Il est nécessaire de passer par un appel de fonction recevant en paramètre le nom du fichier et retournant, dans le cas des fichiers non bufferisés, un descripteur de fichier (entier positif).

Il en est de même pour une file de messages (et également pour les mémoires partagées et les sémaphores). A l'aide de la **clé commune partagée**, on peut obtenir un **entier positif appelé identifiant** (ou **identificateur**) de la file de messages. C'est à l'aide de cet identifiant qu'un programme va pouvoir manipuler une file de messages.

Schématiquement, voici l'exemple d'une file de messages de clé 0x62d, d'identifiant 7, utilisée par deux processus indépendants :



Les deux processus doivent posséder une **variable de type int** (appelée **id** sur le schéma) qui contiendra l'identifiant de la file de message. C'est à l'aide de leur variable **id** qu'ils pourront envoyer et lire des messages sur la file de messages.

6.3.5 Obtention d'une file de messages : la fonction `msgget`

L'obtention d'une file de messages se fait à l'aide de l'appel système **msgget** qui permet, selon le cas, de

1. **créer** une file de messages et d'**obtenir son identifiant**,
2. **obtenir l'identifiant** d'une file de messages qui **existe déjà**.

C'est donc la même fonction qui permet à un processus de créer une nouvelle file de messages ou de récupérer l'identifiant d'une file de messages qui a été créée par un autre processus.

Le prototype de cette fonction est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

où

- **key** est la clé de la file de messages que l'on désire créer, ou de la file de messages existante dont on veut récupérer l'identifiant,
- **msgflg** est un paramètre entier permettant de préciser si l'on souhaite créer une nouvelle file de messages ou simplement récupérer l'identifiant d'une file de messages existante.

Plusieurs combinaisons de paramètres sont possibles :

key	msgflg	action
IPC_PRIVATE		Création d'une file de messages dont on laisse le système choisir la clé. Cela est suffisant lorsque la file de messages est utilisée uniquement par des processus parents (qui héritent de l'identifiant de la file de messages sans devoir faire un nouvel appel à la fonction msgget).
> 0	IPC_CREAT droits	Création d'une file de messages dont la clé est précisée par key . Les droits correspondent aux permissions de lecture/écriture/exécution pour le propriétaire, le groupe et les autres (exactement comme cela est fait pour les fichiers avec la fonction open).
> 0	= 0	Récupération (sans création) de l'identifiant de la file de messages dont la clé est key .

On peut également combiner (par un OU bit à bit) **IPC_CREAT** et **IPC_EXCL**. Dans ce cas, si l'on souhaite créer une file de message dont la **clé est déjà utilisée** par une autre file de messages existante, cela provoquera une **erreur** au lieu d'**écraser** la file de messages existante. Ceci est similaire au comportement de la fonction open avec la combinaison **O_CREAT | O_EXCL**.

Lors de la **création** de la file de messages (avec **key > 0**) :

- le **créateur** et **propriétaire** (structure ipc_perm) sont initialisés avec le UID du processus appelant. Il en est de même pour le groupe propriétaire et le groupe créateur.
- le champ **mode** de la structure ipc_perm est initialisé avec les droits demandés.
- Les champs **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime** et **msg_rtime** de la structure msqid_ds sont initialisés à 0.

- Le champ **msg_ctime** de la structure msqid_ds est rempli avec l'heure actuelle.
- Le champ **msg_qbytes** de la structure msqid_ds est rempli avec la limite système MS-GMNB.

Valeurs de retour

Cette fonction retourne

- l'**identifiant de la file de messages** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EEXIST	Une file de message existe déjà pour cette clé key et IPC_CREAT IPC_EXCL est spécifié
EACCES	Une file de messages existe déjà pour cette clé key, mais le processus appelant n'a pas les permissions pour y accéder.
ENOENT	Aucune file de messages n'existe associée à la clé key et msgflg ne contient pas IPC_CREAT.
ENOSPC	Le nombre maximal de files de messages sur le système est atteint.
...	...

Exemple 6.3 (msgget pour une création). Soit le programme suivant :

```
***** CreationFileMessages.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;

    cle = atoi(argv[1]);

    if ((idQ = msgget(cle,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        perror("Erreur de msgget");
        exit(1);
    }

    printf("idQ = %d\n",idQ);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages

# ./CreationFileMessages 1234
idQ = 98304
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 98304 student      600      0

# ./CreationFileMessages 1234
Erreur de msgget: File exists
#
```

On constate que :

1. Ce programme permet de **créer une file de messages** dont la clé est précisée en **premier argument** de la ligne de commande. La clé est donc récupérée dans argv[1] sous forme d'une chaîne de caractères puis convertie et stockée sous forme de **key_t** dans la variable **cle**.
2. Lors de la première exécution, la file de messages est créé correctement. La **clé 0x4d2** apparaissant lors de l'appel de la commande **ipcs** correspond à la **valeur hexadécimale de 1234**.
3. L'**identifiant 98304** a été attribué à la file de messages lors de l'appel système **msgget** et stocké dans la variable **idQ**.
4. Les **droits** associés à la file de messages sont **600**, c'est-à-dire un accès en **lecture/écriture pour le propriétaire** et rien pour les autres. Les processus dont le UID est le propriétaire de la file de messages pourront dès lors envoyer et lire des messages sur la file de messages.
5. Lors de la seconde exécution, on a voulu créer une seconde file de messages avec la **même clé 1234**, ce qui a provoqué une **erreur** vu que cette clé est déjà utilisée par une file de messages existante et que **IPC_CREAT | IPC_EXCL** a été spécifié dans l'appel de **msgget**.



Exemple 6.4 (msgget pour une récupération). Soit le programme suivant :

```
***** RecupFileMessages.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;

    cle = atoi(argv[1]);
```

```

if ((idQ = msgget(cle,0)) == -1)
{
    perror("Erreur de msgget");
    exit(1);
}

printf("idQ = %d\n",idQ);
exit(0);
}

```

dont un exemple d'exécution fournit

```

# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 98304 student       600      0

# ./RecupFileMessages 1111
Erreur de msgget: No such file or directory
# ./RecupFileMessages 1234
idQ = 98304
#

```

On constate que :

1. Ce programme permet de **récupérer l'identifiant** d'une file de messages dont la clé est précisée en **premier argument** de la ligne de commande. Comme dans l'exemple précédent, la clé est donc récupérée dans argv[1] sous forme d'une chaîne de caractères puis convertie et stockée sous forme de **key_t** dans la variable **cle**.
2. La première exécution a provoqué une **erreur** car on a tenté de récupérer l'identifiant de la file de messages de **clé 1111** qui n'existe pas.
3. La seconde exécution a permis de récupérer l'**identifiant 98304** de la file de messages existante de clé **1234**.



6.3.6 Suppression d'une file de messages : la fonction msgctl

Une application ayant alloué des ressources, comme une file de messages par exemple, se doit de supprimer ces ressources une fois qu'elle se termine, et cela sans que l'utilisateur doive recourir manuellement à la commande **ipcrm** mais bien par des **appels systèmes** dédiés.

L'appel système qui permet de **supprimer** une file de messages est :

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

```

où

- **msqid** est l'**identifiant** de la file de message que l'on désire manipuler,
- **cmd** est un paramètre entier permettant de spécifier la **commande** (c'est-à-dire l'**action**) à réaliser sur la file de messages,
- **buf** est l'adresse d'une structure de type **msqid_ds** locale au processus appelant et qui lui permet de récupérer en local ou de modifier certains paramètres de la file de messages identifiée par msqid.

Voici quelques commandes/actions possibles :

cmd	buf	action
IPC_RMID	NULL	Suppression de la file de messages d'identifiant msqid .
IPC_STAT	≠NULL	Les informations de l'entrée correspondant à l'identifiant msqid de la table des files de messages sont récupérées en local dans la structure msqid_ds pointée par buf.
IPC_SET	≠NULL	Les informations de l'entrée correspondant à l'identifiant msqid de la table des files de messages sont modifiées par celles contenues dans la structure msqid_ds pointée par buf. Seuls les champs suivants de la structure peuvent être mis à jour : msg_qbytes, msg_perm.uid, msg_perm.gid et les 9 bits de poids faible de msg_perm.mode.

Comme on peut le remarquer, la fonction **msgctl** permet bien plus que de simplement supprimer une file de messages. Elle permet un "**contrôle**" de la file de messages, comme par exemple en modifier le propriétaire ou encore les droits d'accès.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	L'argument cmd vaut IPC_STAT, mais le processus appelant n'a pas d'accès en lecture sur la file de messages d'identifiant msqid .
EINVAL	cmd ou msqid ont une valeur illégale.
EFAULT	L'argument cmd vaut IPC_SET ou IPC_STAT mais buf pointe en dehors de l'espace d'adressage autorisé.
EPERM	L'argument cmd vaut IPC_SET ou IPC_RMID mais l'UID effectif du processus appelant n'est pas le créateur (comme indiqué dans msg_perm.cuid) ou le propriétaire (comme indiqué dans msg_perm.uid) de la file de messages.
...	...

Exemple 6.5 (msgctl). Soit le programme suivant :

```
***** TestMsgctl.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;
    msqid_ds file;

    cle = atoi(argv[1]);

    if ((idQ = msgget(cle,0)) == -1)
    {
        perror("Erreur de msgget");
        exit(1);
    }

    printf("idQ = %d\n",idQ) ;

    // Recuperation en local de l'entree dans la table des files de messages
    if (msgctl(idQ,IPC_STAT,&file) == -1)
    {
        perror("Erreur de msgctl(1)");
        exit(1);
    }

    printf("Proprietaire : %d (groupe %d)\n",file.msg_perm.uid,file.msg_perm.gid);
    printf("Droits : 0%o\n",file.msg_perm.mode);
    printf("Nb Messages : %d\n",file.msg_qnum);

    // Suppression de la file de messages
    if (msgctl(idQ,IPC_RMID,NULL) == -1)
    {
        perror("Erreur de msgctl(2)");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 98304 student      600      0

# ./TestMsgctl 1234
idQ = 98304
Proprietaire : 1000 (groupe 1000)
Droits : 0600
Nb Messages : 0
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
#
```

On constate que :

1. Ce programme récupère tout d'abord l'identifiant de la file de messages dont la clé est précisée en premier argument de la ligne de commande.
2. Le premier appel de **msgctl** avec cmd = **IPC_STAT** permet de récupérer en local dans la variable **file** la structure **msqid_ds** de la file de messages. On en affiche ici le propriétaire (student = 1000), le groupe propriétaire (student = 1000), les droits (600) et le nombre de messages présents actuellement (0).
3. Le second appel à **msgctl** avec cmd = **IPC_RMID** n'utilise pas le 3ème paramètre (qui est **NULL**) et a **supprimé** la file de message.



6.3.7 Structure d'un message

Un message doit obligatoirement être une structure mais le contenu peut être tout à fait arbitraire et choisi par le programmeur. Tout message doit avoir la forme suivante :

```
struct XXX
{
    long type;
    // Charge utile du message
    ...
};
```

où

- **XXX** est le nom de la structure. Celui-ci peut être choisi librement par le programmeur et un **typedef** peut également être réalisé.

- **type** le **type du message, obligatoirement présent en premier champ**, de type **long** et > **0**. Le nom de ce champ peut également être choisi arbitrairement par le programmeur.
- ... est la **charge utile** du message (le “**contenu de l’enveloppe**”). Il peut être composé d’un ou plusieurs champs pouvant être choisis par le programmeur.

Les seules **contraintes sur un message** sont donc

1. il doit être une **structure**,
2. le **premier champ** de cette structure doit être un **long**.

Un exemple de **charge utile** pourrait être :

```
char nom[20];
float salaire;
int nbEnfants;
```

et dans ce cas, la structure à utiliser pourrait être

```
struct Message
{
    long type;
    char nom[20];
    float salaire;
    int nbEnfants;
};
```

et la taille du message, ou de la charge utile, est égale à

```
sizeof(struct Message) - sizeof(long)
```

car, comme déjà mentionné, la taille du message ne prend pas compte le type du message (long).

6.3.8 Envoyer un message : la fonction `msgsnd`

La fonction qui permet d’envoyer un message sur une file de message est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

où

- **msqid** est l'**identifiant** de la file de messages sur laquelle on désire envoyer le message,
- **msgp** est l'**adresse du message** (c'est-à-dire un pointeur sur la structure du message),
- **msgsz** est la **taille du message**, c'est-à-dire la taille de la charge utile telle que décrite ci-dessus (**taille de la structure sans le type**),
- **msgflg** est un paramètre entier permettant de **modifier l’“envoi standard”** du message.

type peut pas être neg sinon erreur ! => Q examen

L' "**envoi standard**" d'un message correspond à **msgflg = 0**. Ceci correspond au fait que la fonction msgsnd est **bloquante** jusqu'au moment où l'envoi du message est possible. Ceci peut arriver si la file de messages est pleine. Pour éviter que le processus attende et rende l'appel système msgsnd **non bloquant**, on donne la valeur **IPC_NOWAIT** au paramètre msgflg.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	Le processus n'a pas les droits d'écriture sur la file de messages.
EINVAL	msqid ou msgsz ont une valeur illégale, ou le type du message n'est pas une valeur entière strictement positive .
EFAULT	msgp est une adresse invalide.
EAGAIN	Le message ne peut être envoyé car la file de messages est pleine et le champ msgflg vaut IPC_NOWAIT .
...	...

Exemple 6.6 (msgsnd). Soit le programme suivant :

```
***** TestMsgsnd.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct Message
{
    long type;
    char txt[20];
    int val;
};

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;
    struct Message M;

    // Vérification des arguments de la ligne de commande
    if (argc != 5)
    {
        printf("Trop peu d'arguments !\n");
        printf("Usage : TestMsgsnd cle type txt val\n\n");
        exit(1);
    }
}
```

```

// Recuperation de l'identifiant de la file de messages
cle = atoi(argv[1]);
if ((idQ = msgget(cle,0)) == -1)
{
    perror("Erreur de msgget");
    exit(1);
}

printf("idQ = %d\n",idQ) ;

// Preparation et envoi du message
M.type = atoi(argv[2]);
strcpy(M.txt,argv[3]);
M.val = atoi(argv[4]);

if (msgsnd(idQ,&M,sizeof(struct Message)-sizeof(long),0) == -1)
{
    perror("Erreur de msgsnd");
    exit(1);
}

printf("Processus %d a envoye un message de type %d avec succes.\n",getpid(),M.type);
exit(0);
}

```

dont un exemple d'exécution fournit

```

# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 163840 student       600     0

# ./TestMsgsnd 1234 -2 hello 25
idQ = 163840
Erreur de msgsnd: Invalid argument
# ./TestMsgsnd 1234 7 hello 25
idQ = 163840
Processus 32073 a envoye un message de type 7 avec succes.
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 163840 student       600    24          1

# ipcs -q -i 163840

File de messages msqid=163840
uid=1000 gid=1000 cuid=1000 cgid=1000 mode=0600
octetsc=24 octetsq=16384 qnum=1 lspid=32073 lrpid=0
send_time=Wed Jul 28 17:33:00 2021
rcv_time=Non initialisé
change_time=Wed Jul 28 17:32:23 2021

#

```

On constate que :

1. Ce programme **envoie un message** sur une file de messages. Pour cela, il récupère tout d'abord sur la ligne de commande la clé de la file de messages, le type du message à envoyer, le contenu de la charge utile du message constituée d'une chaîne de caractères et d'un entier. Notez que le nombre d'arguments est contrôlé en testant le paramètre **argc** de la fonction main.
2. La première exécution du programme provoque une **erreur** car on a tenté d'envoyer un message dont le **type n'est pas valide** (valeur entière négative (-2)).
3. Le message envoyé par le programme est représenté par la variable **M (struct Message)**. La **charge utile** du message est donc une chaîne de caractères **txt** (char[20]) et un entier **val** (int, 4 octets).
4. Après la seconde exécution du programme qui, elle, a réussi, la file de message d'identifiant 163840 contient un message. Ceci est vérifié par l'exécution de la commande **ipcs**.
5. On remarque que la file de message contient **24** octets. Ceci correspond à la **taille de la charge utile** du message (20 octets pour char[20] et 4 octets pour l'entier).
6. L'utilisation de la commande **ipcs** avec l'option **-i** permet de visualiser **tous les détails** des informations concernant la file de messages. On retrouve évidemment le nombre de messages présents (**qnum = 1**) et le nombre d'octets présents sur la file de message (**octetsc = 24**) mais également le **PID** du dernier processus qui a émis un message sur la file de messages (**lspid = 32073**). Ceci est particulièrement pratique pour mettre au point un programme en contrôlant qui a envoyé quoi sur une file de messages.



6.3.9 Recevoir un message : la fonction `msgrecv`

La fonction qui permet de lire un message sur une file de messages est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrecv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

où

- **msqid** est l'**identifiant** de la file de messages sur laquelle on désire lire un message,
- **msgp** est l'**adresse du buffer de réception**, c'est-à-dire d'une variable locale au processus et qui contiendra le message lu de la file de messages,
- **msgsz** est la **taille du buffer de réception**, c'est-à-dire la taille de la variable qui va accueillir le message lu sur la file de messages. Il est important que la taille du buffer de réception soit au moins égale à la taille du message à lire. D'une manière imagée, il faut que la boîte aux lettres de réception puisse "contenir le colis reçus". De nouveau, msgsz ne doit **pas tenir compte du type du message**.

- **msgtyp** est un paramètre entier qui permet de choisir le **type de message** que l'on désire lire sur la file de messages :
 - **msgtyp = 0** : le plus ancien quel que soit son type,
 - **msgtyp > 0** : le plus ancien dont le type est **égal à msgtyp**,
 - **msgtyp < 0** : le plus ancien ayant le plus petit type inférieur ou égal à la valeur absolue de msgtyp. **blc celui là xD**
- **msgflg** est un paramètre entier permettant de **modifier la "réception standard"** d'un message.

La **réception standard** d'un message correspond à **msgflg = 0**. Ceci correspond au fait que la fonction msgrcv est **bloquante** jusqu'au moment où un message de type désiré apparaît sur la file de messages. Pour éviter que le processus attende et rende l'appel système msgrcv **non bloquant**, on donne la valeur **IPC_NOWAIT** au paramètre msgflg. D'autres valeurs sont possibles pour ce paramètre et seront abordées plus loin.

Choisir **msgtyp = 0** revient à ne pas utiliser le type des messages. Dans ce cas, la file de messages se comporte comme une simple liste FIFO.

La lecture est **destructrice**, c'est-à-dire qu'une fois lu, un message disparaît de la file de messages. Un message ne peut donc être lu que par un seul processus. Si on veut envoyer un même message à plusieurs processus, il sera donc nécessaire de l'envoyer en autant d'exemplaires.

Valeurs de retour

Cette fonction retourne

- la **taille de la charge utile (pas de la structure complète !) du message lu** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EIDRM	La file de messages a été supprimée pendant que le processus était endormi sur msgrcv.
EINVAL	msqid ou msgsz ont une valeur illégale.
EFAULT	msgp est une adresse invalide.
ENOMSG	Il n'y a pas de message de type désiré sur la file de messages et le champ msgflg vaut IPC_NOWAIT .
E2BIG	La taille du message à lire est plus grande que msgsz et le flag msgflg n'est pas positionné à MSG_NOERROR (voir plus loin). Dans ce cas, le message n'est pas lu et reste présent sur la file de messages.
EACCES	Le processus n'a pas les droits en lecture sur la file de messages.
...	...

Exemple 6.7 (msgrcv). Soit le programme suivant :

```
***** TestMsgrcv.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct Message
{
    long type;
    char txt[20];
    int val;
};

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;
    struct Message M;
    long type;
    int ret;

    // Verification des arguments de la ligne de commande
    if (argc != 3)
    {
        printf("Trop peu d'arguments !\n");
        printf("Usage : TestMsgrcv cle type\n\n");
        exit(1);
    }

    // Recuperation de l'identifiant de la file de messages
    cle = atoi(argv[1]);
    if ((idQ = msgget(cle,0)) == -1)
    {
        perror("Erreur de msgget");
        exit(1);
    }

    printf("idQ = %d\n",idQ);

    // Lecture d'un message
    type = atoi(argv[2]);

    if ((ret = msgrcv(idQ,&M,sizeof(struct Message)-sizeof(long),type,0)) == -1)
    {
        perror("Erreur de msgrcv");
        exit(1);
    }

    printf("Processus %d a lu un message avec succes.\n",getpid());
    printf("type = %d\n",M.type);
    printf("ret = %d\n",ret);
    printf("txt = %s\n",M.txt);
    printf("val = %d\n",M.val);
    exit(0); }
```

dont un exemple d'exécution fournit

```
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 327680 student       600    0

# ./TestMsgsnd 1234 5 hello 34
idQ = 327680
Processus 2566 a envoyé un message de type 5 avec succès.
# ./TestMsgsnd 1234 8 coucou 18
idQ = 327680
Processus 2574 a envoyé un message de type 8 avec succès.
# ./TestMsgsnd 1234 2 bye 21
idQ = 327680
Processus 2581 a envoyé un message de type 2 avec succès.
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 327680 student       600    72           3

# ./TestMsgrcv 1234 0
idQ = 327680
Processus 2603 a lu un message avec succès.
type = 5
ret = 24
txt = hello
val = 34
# ./TestMsgrcv 1234 2
idQ = 327680
Processus 2611 a lu un message avec succès.
type = 2
ret = 24
txt = bye
val = 21
# ipcs -q -i 327680

File de messages msqid=327680
uid=1000 gid=1000 cuid=1000 cgid=1000 mode=0600
octetsc=24 octetsq=16384 qnum=1 lspid=2581 lrpid=2611
send_time=Thu Jul 29 08:39:05 2021
rcv_time=Thu Jul 29 08:39:33 2021
change_time=Thu Jul 29 08:38:36 2021

# ./TestMsgrcv 1234 9
idQ = 327680
```

On constate que :

1. Tout d'abord, 3 messages de types différents (5, 8 et 2) ont été envoyés sur la file de messages en utilisant le programme **TestMsgsnd** de l'**exemple 6.6**.
2. Le programme TestMsgrcv **lit un message** sur une file de messages. Pour cela, il récupère

tout d'abord sur la ligne de commande la clé de la file de messages et le type désiré d'un message à lire. Notez que le nombre d'arguments est contrôlé en testant le paramètre **argc** de la fonction main.

3. La variable **M** fait ici office de **boîte au lettres de réception**. Elle dispose d'une taille identique au message qui va être lu. Le paramètre **msgsz** passé à la fonction **msgrcv** correspond à la taille de la structure de M mais **sans le type**.
4. La **première exécution** du programme lit le **message le plus ancien** sur la file de message (celui de type 5). En effet, dans ce cas, la paramètre **msgtyp** passé à la fonction **msgrcv** est égal à **0**.
5. La **seconde exécution** du programme lit le **message de type 2** comme précisé par le paramètre **msgtyp** de la fonction **msgrcv**.
6. Le **retour** de la fonction **msgrcv** lors des deux premières exécutions est égal à **24**, ce qui correspond bien à la **taille de la charge utile** du message.
7. Lors de l'exécution de commande **ipcs** avec l'option **-i**, on observe qu'il ne reste qu'un message sur la file de messages. Les deux messages lus précédemment ont donc bien été **supprimés de la file** de messages. De plus, on observe les **PID** des derniers processus qui ont émis (2581) et lu (2611) un message sur la file de messages.
8. La **dernière exécution** du programme est restée **bloquée sur l'appel système msgrcv** car le type de message souhaité est 9 alors que le seul message encore présent sur la file de messages est de type 8. Nous sommes donc bien dans une situation de réception standard.



6.3.10 Quelques variantes à la réception standard d'un message

Le paramètre **msgflg** permet de modifier le comportement de la fonction **msgrcv**. Il peut, par exemple, prendre les valeurs

A UTILISER **IPC_NOWAIT** : dans ce cas, la fonction **msgrcv n'est plus bloquante** s'il n'y a pas de message de type désiré présent sur la file de messages. La fonction **msgrcv** retourne alors **-1** et **errno** est positionné **ENOMSG**. Ceci est particulièrement pratique si on désire qu'un processus ne reste pas continuellement bloqué sur la réception d'un message alors qu'il pourrait réaliser d'autres tâches.

- pas fort utilisé*
- **MSG_NOERROR** : dans ce cas, si le message à lire dans la file de messages a une **charge utile de taille supérieure à la celle du buffer de réception** (paramètre **msgsz** de **msgrcv**), le message est **lu malgré tout** mais est **tronqué** à **msgsz** octets sans provoquer d'erreur.
 - **MSG_EXCEPT** : dans ce cas, la fonction **msgrcv** va lire un message de n'importe quel type **sauf** celui précisé par le paramètre **msgtyp**.

Le champs **msgflg** peut également être une **combinaison** de plusieurs de ces constantes en utilisant l'opérateur OU logique **|**.

Exemple 6.8 (msgflg). Soit le programme suivant :

```
***** TestMsgflg.cpp *****
#include <stdio.h>
```

bourrin/ pas utilisé / en gros ça coupe et blc du reste

paquet facteur coupé

```

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

struct Message
{
    long type;
    char txt[20];
    int val;
};

int main(int argc,char* argv[])
{
    int idQ;
    key_t cle;
    struct Message M;
    int ret;

    // Recuperation de l'identifiant de la file de messages
    cle = atoi(argv[1]);
    if ((idQ = msgget(cle,0)) == -1)
    {
        perror("Erreur de msgget");
        exit(1);
    }

    printf("idQ = %d\n",idQ);

    // Lecture d'un message de type 2 (non present)
    if ((ret = msgrcv(idQ,&M,sizeof(struct Message)-sizeof(long),2,IPC_NOWAIT)) == -1)
        perror("Erreur de msgrcv (1)");

    // Lecture d'un message de type 1 (present) trop long
    errno = 0;
    if ((ret = msgrcv(idQ,&M,5,1,0)) == -1)
        perror("Erreur de msgrcv (2)");

    // Lecture d'un message de type 1 (present) trop long mais tronqué
    errno = 0;
    if ((ret = msgrcv(idQ,&M,5,1,MSG_NOERROR)) == -1)
        perror("Erreur de msgrcv (3)");

    printf("Processus %d a lu un message avec succès.\n",getpid());
    printf("type = %d\n",M.type);
    printf("ret = %d\n",ret);
    printf("txt = %s\n",M.txt);
    printf("val = %d\n",M.val);
    exit(0); }
```

dont un exemple d'exécution fournit

```
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 360448 student       600    0           0

# ./TestMsgsnd 1234 1 abcdefghij 34
idQ = 360448
Processus 4134 a envoyé un message de type 1 avec succès.
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 360448 student       600   24           1

# ./TestMsgflg 1234
idQ = 360448
Erreur de msgrcv (1): No message of desired type
Erreur de msgrcv (2): Argument list too long
Processus 4241 a lu un message avec succès.
type = 1
ret = 5
txt = abcde
val = 0
#
```

On constate que :

1. Tout d'abord, un message de **type 1** a été envoyé sur la file de messages en utilisant le programme **TestMsgsnd** de l'**exemple 6.6**.
2. Lors du **premier appel** de la fonction de msgrcv (avec **msgflg = IPC_NOWAIT**), le type désiré d'un message à lire est égal à **2**. Bien qu'il n'y ait pas de message de type 2 sur la file de messages, la fonction **msgrcv n'est pas bloquante** et a retourné **-1**. Ceci est confirmé par "**No Message of desired type**".
3. Lors du **second appel** à msgrcv, le type désiré de message à lire correspond à celui du message présent sur la file de messages. Cependant le message n'est pas lu. Ceci est dû au fait que la **taille de charge utile du message est de 24 octets** tandis que la fonction msgrcv a reçu en paramètre une **taille de buffer de réception égale à 5**, ce qui est **insuffisant** pour recevoir le message. Celui-ci n'est donc **pas lu** et la fonction msgrcv a retourné **-1**. Ceci est confirmé par "**Argument list too long**".
4. Lors du **dernier appel** de la fonction msgrcv (avec **msgflg = MSG_NOERROR**), le message a été **lu sans erreur** bien que la taille 24 de sa charge utile soit supérieure à 5. Cependant, le message a été **tronqué**, comme l'indique la variable **M.txt** qui contient "**abcde**" au lieu de "**abcdefghij**".



6.3.11 Application : Client/Serveur mono-processus

Nous allons illustrer ici l'utilisation d'une file de messages afin de créer une petite **application client/serveur** de calcul très simple :

finale => multi processus, père attend et fils fais les taches

1. Le programme **Serveur** va créer un file de messages et se mettre, **en boucle**, en attente d'une requête en provenance d'un programme **Client**.
2. Le programme **Client** va lire en ligne de commande une opération mathématique simple, soit une addition comme par exemple "3 + 5", soit une soustraction "11 - 3" et **envoyer une requête** de calcul **au processus Serveur**.
3. Le processus **Serveur** va recevoir, **traiter la requête** et **envoyer la réponse** au **bon processus Client**.

Dans cette application, le **type d'un message** va correspondre au **PID du destinataire** du message. Le serveur pourra assez facilement connaître le PID du client qui a envoyé le message (il suffit de **placer de le PID de l'expéditeur dans la charge utile du message**) et donc affecter correctement le type du message faisant office de réponse. Par contre, il est **impossible à priori** au client de **connaître le PID du serveur** (celui-ci change en plus à chaque fois que le serveur est relancé). La **convention** est alors de fixer à la **valeur 1 le type de message dont le destinataire est le serveur**. Dès lors,

- le **serveur** se mettra en **attente** de réception de **messages de type 1**, tandis que
- le **client** se mettra en **attente** de **messages de type correspondant à son propre PID**.

Un fichier header **protocole.h** inclus par les deux programmes **Serveur.cpp** et **Client.cpp** va également être utilisé pour mettre en place le "protocole" de communication entre les processus, à savoir :

1. la **clé partagée** pour la file de messages,
2. les **structures REQUETE et REPONSE** qui sont utilisées par le serveur et le client.

Exemple 6.9 (Client/Serveur). Voici tout d'abord le fichier protocole.h partagé par le serveur et le client :

```
***** protocole.h *****
#ifndef PROTOCOLE_H
#define PROTOCOLE_H

#define CLE 1234

typedef struct
{
    long type;
    pid_t expediteur;
    char op;
    int val1;
    int val2;
} REQUETE;

typedef struct
{
    long type;
    int resultat;
} REPONSE;

#endif
```

La **charge utile** d'une requête est donc composée du **PID de l'expéditeur**, des deux opérandes **val1** et **val2** et de l'opération **op** à réaliser sous forme d'un simple caractère (char) pouvant

prendre les valeurs '+' ou '-' dans cet exemple simple. La charge utile de la réponse correspond simplement au **résultat** du calcul.

Voici à présent le code du serveur :

```
WHEW      ( code non complet, il faut aussi que le serveur se finisse
            correctement avec une fermeture des message grace à
            l'handler )
```

```
***** Serveur.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "protocole.h"

int main(int argc,char* argv[])
{
    int idQ;
    REQUETE req;
    REPONSE rep;

    // Creation de la file de messages
    if ((idQ = msgget(CLE,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        perror("(Serveur) Erreur de msgget");
        exit(1);
    }

    printf("(Serveur) demarre (%d)... \n",CLE);

    while(1)
    {
        // Attente d'une requete          le serveur attend un message de type 1 ( convention )
        if (msgrecv(idQ,&req,sizeof(REQUETE)-sizeof(long),1,0) == -1)
        {
            adresse requete ( boite au lettre )
            perror("(Serveur) Erreur de msgrecv");
            msgctl(idQ,IPC_RMID,NULL);
            exit(1);
        }

        printf("(Serveur) Requete recue de %d\n",req.expediteur);

        // Traitement de la requete
        printf("(Serveur) Traitement de la requete : %d %c %d\n",req.val1,req.op,req.val2);
        switch (req.op)
        {
            case '+': rep.resultat = req.val1 + req.val2;
                        break;

            case '-': rep.resultat = req.val1 - req.val2;
                        break;
        }

        // Preparation et envoi de la reponse
        rep.type = req.expediteur; // destinataire de la reponse = expediteur de la requete
        if (msgsnd(idQ,&rep,sizeof(REPONSE)-sizeof(long),0) == -1)
        {
            perror("(Serveur) Erreur de msgsnd");
            msgctl(idQ,IPC_RMID,NULL);
            exit(1);
        }
    }
}
```

A retenir

```

    }
    exit(0);
}

```

On constate que :

1. Si la réception de la requête ou l'envoi de la réponse échoue, le serveur **supprime la file de messages** avant de se terminer (utilisation de **msgctl** avec **IPC_RMID** en paramètre). Le serveur, ayant créé la resource système, se doit de la supprimer.
2. Comme dit plus haut, le **type de message attendu par le serveur est 1 par convention**.
3. la ligne de code **rep.type = req.expediteur;** est très importante. C'est elle qui va "**router la réponse** vers le **bon processus client**.

Voici à présent le code du client :

```

***** Client.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "protocole.h"

int main(int argc,char* argv[])
{
    int idQ;
    REQUETE req;
    REPONSE rep;

    // Verification des arguments de la ligne de commande
    if (argc != 4)
    {
        printf("Trop peu d'arguments !\n");
        printf("Usage : Client valeur1 operation valeur2\n\n");
        exit(1);
    }

    // Recuperation de l'identifiant de la file de messages
    if ((idQ = msgget(CLE,0)) == -1)
    {
        car je ne crée pas 0
        perror("(Client) Erreur de msgget");
        exit(1);
    }

    // Preparation et envoi de la requete
    req.type = 1; // destinataire : 1 est une convention pour le serveur
    req.expediteur = getpid();
    req.val1 = atoi(argv[1]);
    req.val2 = atoi(argv[3]);
    req.op = argv[2][0];

    if (msgsnd(idQ,&req,sizeof(REQUETE)-sizeof(long),0) == -1)
    {
        perror("(Client) Erreur de msgsnd");
        exit(1);
    }
}

```

```

printf("(Client %d) Envoi requete, attente reponse... \n",getpid());

// Attente de la reponse
if (msgrecv(idQ,&rep,sizeof(REPONSE)-sizeof(long),getpid(),0) == -1)
{
    perror("(Client) Erreur de msgrecv");
    exit(1);
}

// Affichage du resultat
printf("(Client %d) Reponse : %d %c %d = %d\n",getpid(),req.val1,req.op,req.val2,rep.resultat);
exit(0);
}

```

On constate que :

1. Le **type** de la requête a été fixé à **1**, dans le but d'envoyer cette requête au processus **serveur**.
2. Le **champ expediteur** de la requête a été affecté avec le **PID de l'expéditeur**, c'est-à-dire du client, à l'aide de la fonction **getpid**.
3. Lors de l'appel à la fonction **msgrecv**, le paramètre correspondant au **type** de message attendu par le client correspond au **PID de ce client**, de nouveau obtenu à l'aide de la fonction **getpid**.

Une exemple d'exécution de cette application client/serveur est fourni par

```

# ipcs -q

----- Files de messages -----
clef msqid propriétaire perms octets utilisés messages

# ./Serveur &
[1] 5997
(Serveur) demarre (1234)...
# ./Client 4 + 11
(Client 6012) Envoi requete, attente reponse...
(Serveur) Requete recue de 6012
(Serveur) Traitement de la requete : 4 + 11
(Client 6012) Reponse : 4 + 11 = 15
# ./Client 23 - 18
(Client 6019) Envoi requete, attente reponse...
(Serveur) Requete recue de 6019
(Serveur) Traitement de la requete : 23 - 18
(Client 6019) Reponse : 23 - 18 = 5
# ipcs -q

----- Files de messages -----
clef      msqid  propriétaire perms octets utilisés messages
0x000004d2 524288 student      600      0

# ipcrm -q 524288
(Serveur) Erreur de msgrecv: Identifier removed
[1]+ Termine 1          ./Serveur
#

```

On constate que :

1. Le **serveur** est lancé en tâche de fond, il se comporte comme un processus **démon** qui attend des requêtes et y répond.
2. La commande **ipcrm** supprime la file de message manuellement. Ceci a pour effet de terminer le processus Serveur. En effet, celui-ci était endormi sur l'appel système **msgrcv** qui a alors retourné **-1**, errno étant positionné à **EIDRM**. Ce n'est pas du tout la meilleure chose à faire. Idéalement, il faudrait **armer un signal (SIGINT idéalement)** pour le processus Serveur sur un **handler** qui **supprimerait proprement et automatiquement la file de messages** avant que le processus ne s'arrête. L'envoi du signal SIGINT au processus serveur l'arrêterait alors proprement tout en supprimant la file de messages.



On parle ici d'un **serveur mono-processus** car le processus Serveur

- attend les requêtes,
- traite lui-même les requêtes,
- envoie lui-même les réponses aux clients,

et cela sans créer de nouveaux processus, il est **seul pour tout faire**. Pendant le traitement d'une requête (ce qui pourrait être long dans le cas d'autres traitements que de simples additions ou soustractions), il est incapable réagir aux nouvelles requêtes qui s'accumulent dans la file de messages en provoquant ainsi un temps d'attente au niveau des clients.

Afin d'éviter ce problème, on pourrait imaginer de créer un **serveur multi-processus** dans lequel

- le processus serveur attendrait les requêtes,
- le processus serveur créerait un nouveau processus (par appel à la fonction **fork** et éventuellement **exec**) à chaque nouvelle requête reçue.
- les processus fils pourraient alors traiter les requêtes et répondre aux clients pendant que le processus serveur se **remettrait** en attente d'une requête sur la fonction **msgrcv**.

6.4 Les mémoires partagées

Comme déjà mentionné dans l'introduction,

- Une mémoire partagée est une **zone de mémoire** allouée (un peu comme un "malloc" le ferait) mais accessible par plusieurs processus différents. Cette zone ne fait pas partie de l'espace adressable d'un processus particulier, elle se situe "**entre les processus**".
- Chaque processus désirant manipuler une mémoire partagée possède **son propre pointeur** vers la zone de mémoire commune. On dit que le processus est **attaché à la mémoire partagée**. Lorsqu'il ne souhaite plus l'utiliser, le processus peut se **détacher** de cette mé-

moire.

- Chaque processus partageant cette mémoire peut y lire ou y écrire **sans appel système** particulier, juste comme un accès mémoire ordinaire. L'**utilisation** pour un processus est donc tout à fait **transparente**, il ne fait pas de différence entre une zone mémoire qu'il a allouée par un malloc ou une mémoire partagée.

Une **table système** est dédiée à la gestion des mémoires partagées.

6.4.1 Structure interne d'une mémoire partagée

La table système des mémoires partagées comprend une entrée distincte pour chaque mémoire partagée et chaque entrée est structurée par la structure suivante :

```
struct shmid_ds
{
    struct ipc_perm shm_perm; /* operation permission struct */

    size_t shm_segsz;        /* size of segment in bytes */

    time_t shm_atime;        /* time of last shmat() */
    time_t shm_dtime;        /* time of last shmdt() */
    time_t shm_ctime;        /* time of last change by shmctl() */

    pid_t  shm_cpid;         /* pid of creator */
    pid_t  shm_lpid;         /* pid of last shmop */

    shmat_t shm_nattch;      /* number of current attaches */
};                                nbre de pointeurs, pas le nbre de processus lié
```

où

- **shm_perm** est la structure commune à tous les IPC, reprenant le créateur, le propriétaire, les droits, ...
- **shm_segsz** est la **taille en octets** de la mémoire partagée.
- **shm_atime** et **shm_dtime** sont respectivement les dates/heures des derniers **attachement** et **détachement** à la mémoire partagée.
- **shm_ctime** est la date/heure de création et/ou dernière modification de paramètres de la mémoire partagée.
- **shm_cpid** et **shm_lpid** sont respectivement les PID des processus qui ont créé la mémoire partagée et qui, en dernier, ont manipulé la mémoire partagée.
- **shm_nattach** est le **nombre** (shmat_t est une redéfinition de unsigned int) **de processus actuellement attachés** à la mémoire partagée.

Faisant partie de la famille des IPC au même titre que les files de messages, une mémoire partagée est caractérisée par

- une **clé** commune partagée et,
- un **identifiant** qui permet à chaque processus disposant de la clé de s'**attacher** et de se **détacher** à la mémoire partagée.

6.4.2 Obtention d'une mémoire partagée : la fonction `shmget`

L'obtention d'une mémoire partagée se fait à l'aide de l'appel système **shmget** qui permet, selon le cas, de

1. **créer** une mémoire partagée et d'**obtenir son identifiant**,
2. **obtenir l'identifiant** d'une mémoire partagée qui **existe déjà**.

C'est donc la même fonction qui permet à un processus de créer une nouvelle mémoire partagée ou de récupérer l'identifiant d'une mémoire partagée qui a été créée par un autre processus.

Le prototype de cette fonction est :

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

où

- **key** est la clé de la mémoire partagée que l'on désire créer, ou de la mémoire partagée existante dont on veut récupérer l'identifiant,
- **size** est la **taille en octets** de la mémoire partagée que l'on désire créer.
- **shmflg** est un paramètre entier permettant de préciser si l'on souhaite créer une nouvelle mémoire partagée ou simplement récupérer l'identifiant d'une mémoire partagée existante.

Plusieurs combinaisons de paramètres sont possibles :

key	size	shmflg	action
IPC_PRIVATE	> 0		Création d'une mémoire partagée dont on laisse le système choisir la clé. Cela est suffisant lorsque la mémoire partagée est utilisée uniquement par des processus parents (qui héritent de l'identifiant de la mémoire partagée sans devoir faire un nouvel appel à la fonction shmget).
> 0	> 0	IPC_CREAT droits	Création d'une mémoire partagée dont la clé est précisée par key et de taille correspondant à size . Les droits correspondent aux permissions de lecture/écriture/exécution pour le propriétaire, le groupe et les autres (exactement comme cela est fait pour les fichiers avec la fonction open).
> 0	= 0	= 0	Récupération (sans création) de l'identifiant de la mémoire partagée dont la clé est key .

On peut également combiner (par un OU bit à bit) **IPC_CREAT** et **IPC_EXCL**. Dans ce cas, si l'on souhaite créer une mémoire partagée dont la **clé est déjà utilisée** par une autre mémoire partagée existante, cela provoquera une **erreur** au lieu d'**écraser** la mémoire partagée existante. Ceci est similaire au comportement de la fonction open avec la combinaison **O_CREAT | O_EXCL**.

Lors de la **création** de la mémoire partagée (avec **key > 0**) :

- le **créateur** et **propriétaire** (structure ipc_perm) sont initialisés avec le UID du processus appelant. Il en est de même pour le groupe propriétaire et le groupe créateur.
- le champ **mode** de la structure ipc_perm est initialisé avec les droits demandés.
- Le champs **shm_segsz** de la structure shmid_ds prend la valeur **size**.
- Le champ **shm_ctime** de la structure shmid_ds est rempli avec l'heure actuelle.
- Le champ **shm_nattch** de la structure shmid_ds est initialisé à **0**.

Valeurs de retour

Cette fonction retourne

- l'**identifiant de la mémoire partagée** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EEXIST	Une mémoire partagée existe déjà pour cette clé key et IPC_CREAT IPC_EXCL est spécifié
EACCES	Une mémoire partagée existe déjà pour cette clé key, mais le processus appelant n'a pas les permissions pour y accéder.
ENOENT	Aucune mémoire partagée n'existe associée à la clé key et shmflg ne contient pas IPC_CREAT.
ENOSPC	Le nombre maximal de mémoires partagées sur le système est atteint ou la taille size demandée est supérieure à la limite autorisée.
...	...

Exemple 6.10 (shmget pour une création). Soit le programme suivant :

```
***** CreationShm.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;
    int taille;

    cle = atoi(argv[1]);
    taille = atoi(argv[2]);

    if ((idShm = shmget(cle,taille,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -m

----- Segment de mémoire partagée -----
clef shmid propriétaire perms octets nattch états

# ./CreationShm 1234 100
idShm = 139984951
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid      propriétaire perms octets nattch états
0x000004d2 139984951 student      600    100    0

# ./CreationShm 1234 200
Erreur de shmget: File exists
# ipcs -m -i 139984951

Mémoire partagée segment shmid=139984951
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600 access_perms=0600
octets=100 lpid=0 cpid=12232 nattch=0
att_time=Non initialisé
det_time=Non initialisé
change_time=Mon Aug 2 10:56:37 2021

#
```

On constate que :

1. Ce programme permet de **créer une mémoire partagée** dont la clé et la taille sont précisées en **premier et second argument** de la ligne de commande.
2. Lors de la première exécution, la mémoire partagée est créé correctement. La **clé 0x4d2** apparaissant lors de l'appel de la commande **ipcs** correspond à la **valeur hexadécimale de 1234**.
3. L'**identifiant 139984951** a été attribué à la mémoire partagée lors de l'appel système **shmget** et stocké dans la variable **idShm**.
4. Les **droits** associés à la mémoire partagée sont **600**, c'est-à-dire un accès en **lecture/écriture pour le propriétaire** et rien pour les autres. Les processus dont le UID est le propriétaire de la mémoire partagée pourront dès lors s'y attacher.
5. Lors de la seconde exécution, on a voulu créer une seconde mémoire partagée avec la **même clé 1234**, ce qui a provoqué une **erreur** vu que cette clé est déjà utilisée par une mémoire partagée existante et que **IPC_CREAT | IPC_EXCL** a été spécifié dans l'appel de **shmget**.
6. La commande **ipcs** (avec ou sans l'option **-i**) a permis d'afficher les informations de la mémoire partagée. On y voit par exemple la **taille de la mémoire partagée (100)** et le **nombre de processus attachés** actuellement (**0**).



Exemple 6.11 (shmget pour une récupération). Soit le programme suivant :

```
***** RecupShm.cpp *****/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char* argv[])
{
    int idShm;
    key_t cle;

    cle = atoi(argv[1]);

    if ((idShm = shmget(cle, 0, 0)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n", idShm);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x000004d2 139984951 student       600    100      0

# ./RecupShm 1234
idShm = 139984951
# ./RecupShm 111
Erreur de shmget: No such file or directory
#
```

On constate que :

1. Ce programme permet de **récupérer l'identifiant** d'une mémoire partagée dont la clé est précisée en **premier argument** de la ligne de commande.
2. La première exécution a permis de récupérer l'**identifiant 139984951** de la mémoire partagée existante de clé **1234**.
3. La seconde exécution a provoqué une **erreur** car on a tenté de récupérer l'identifiant de la mémoire partagée de **clé 111** qui n'existe pas.



6.4.3 Suppression d'une mémoire partagée : la fonction shmctl

Comme déjà mentionné, une application ayant alloué des ressources, comme une file de messages ou une mémoire partagée, se doit de supprimer ces ressources une fois qu'elle se termine, et cela sans que l'utilisateur doive recourir manuellement à la commande **ipcrm** mais bien par des **appels systèmes** dédiés.

L'appel système qui permet de **supprimer** une mémoire partagée est :

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

où

- **shmid** est l'**identifiant** de la mémoire partagée que l'on désire manipuler,
- **cmd** est un paramètre entier permettant de spécifier la **commande** (c'est-à-dire l'**action**) à réaliser sur la mémoire partagée,
- **buf** est l'adresse d'une structure de type **shmid_ds** locale au processus appelant et qui lui permet de récupérer en local ou de modifier certains paramètres de la mémoire partagée identifiée par shmid.

Voici quelques commandes/actions possibles :

cmd	buf	action
IPC_RMID	NULL	Suppression de la mémoire partagée d'identifiant shmid .
IPC_STAT	≠NULL	Les informations de l'entrée correspondant à l'identifiant shmid de la table des mémoires partagées sont récupérées en local dans la structure shmid_ds pointée par buf.
IPC_SET	≠NULL	Les informations de l'entrée correspondant à l'identifiant shmid de la table des mémoires partagées sont modifiées par celles contenues dans la structure shmid_ds pointée par buf. Seuls les champs suivants de la structure peuvent être mis à jour : shm_perm.uid, shm_perm.gid et les 9 bits de poids faible de shm_perm.mode.

Comme on peut le remarquer, la fonction **shmctl** permet bien plus que de simplement supprimer une mémoire partagée. Elle permet un "**contrôle**" de la mémoire partagée, comme par exemple en modifier le propriétaire ou encore les droits d'accès.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	L'argument cmd vaut IPC_STAT, mais le processus appelant n'a pas d'accès en lecture sur la mémoire partagée d'identifiant shmid .
EINVAL	cmd ou shmid ont une valeur illégale.
EFAULT	L'argument cmd vaut IPC_SET ou IPC_STAT mais buf pointe en dehors de l'espace d'adressage autorisé.
EPERM	L'argument cmd vaut IPC_SET ou IPC_RMID mais l'UID effectif du processus appelant n'est pas le créateur (comme indiqué dans <code>shm_perm.cuid</code>) ou le propriétaire (comme indiqué dans <code>shm_perm.uid</code>) de la mémoire partagée.
...	...

Exemple 6.12 (shmctl pour une suppression). Soit le programme suivant :

```
***** SuppressionShm.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;

    cle = atoi(argv[1]);

    if ((idShm = shmget(cle,0,0)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm) ;

    // Suppression de la memoire partagee
    if (shmctl(idShm,IPC_RMID,NULL) == -1)
    {
        perror("Erreur de shmctl");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x0000004d2 147587127 student       600    100      1
0x0000004d3 147619930 student       600    150      0

# ./SuppressionShm 1234
idShm = 147587127
# ./SuppressionShm 1235
idShm = 147619930
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x000000000 147587127 student       600    100      1      dest

# ipcrm -m 147587127
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x000000000 147587127 student       600    100      1      dest

#
```

On constate que :

1. Avant exécution du programme, **deux mémoires partagées** existent sur le système, une ayant déjà **un processus qui y est attaché (nattch = 1)** et l'autre n'ayant **aucun processus attaché (nattch = 0)**.
2. Ce programme récupère tout d'abord l'identifiant de la mémoire partagée dont la clé est précisée en premier argument de la ligne de commande pour ensuite tenter de la **supprimer** grâce à la fonction **shmctl** avec le paramètre **IPC_RMID**.
3. La **première exécution** n'a **pas réussi à supprimer** la mémoire partagée et la fonction **shmctl** n'a retournée **aucune erreur**. Une **demande de suppression** a été réalisée, cela se voit grâce à la commande **ipcrm** où on voit apparaître l'état **dest ("destruction")** au niveau de la mémoire partagée en question.
4. La **seconde exécution** a bel et bien **supprimé** la mémoire partagée.



Remarques importantes

1. L'exécution de la commande **ipcrm** ou l'appel à la fonction **shmctl** pour **supprimer une mémoire partagée** ne provoque **pas la suppression immédiate** de la mémoire partagée, une **demande de destruction** est **enregistrée**.
2. Pour qu'une mémoire partagée soit **effectivement supprimée**, il faut que **tous les processus qui y sont attachés s'en détachent ou se terminent**.

6.4.4 Attacher un processus à une mémoire partagée : la fonction shmat

Pour utiliser une mémoire partagée, un processus doit posséder une adresse vers cette zone de mémoire. A partir de l'identifiant obtenu à l'aide de la fonction `shmget`, un processus peut **s'attacher à une mémoire partagée**, c'est-à-dire **obtenir un pointeur vers cette mémoire partagée**, en utilisant la fonction :

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

où

- **shmid** est l'**identifiant** de la mémoire partagée à laquelle le processus appelant souhaite s'attacher,
- **shmaddr** est l'adresse désirée pour l'attachement. Si cette adresse est **NULL**, le noyau recherche un emplacement libre dans l'espace adressable du processus. C'est bien sûr le **mécanisme que l'on utilisera toujours**. Choisir soi-même une adresse d'attachement ne se justifie que dans des cas exceptionnels qui ne nous concernent pas ici.
- **shmflg** est un paramètre entier permettant de préciser la manière dont on désire s'attacher à la mémoire partagée. Ce paramètre peut prendre les valeurs :
 - **SHM_RDONLY** si on désire un attachement en **lecture seule**,
 - **0** si on désire un attachement en **lecture/écriture**.

Un attachement en écriture seule n'est pas possible.

Valeurs de retour

Cette fonction retourne

- l'**adresse vers le début de zone de mémoire partagée** en cas de succès,
- la valeur **(void*)-1** en cas d'erreur, et `errno` est positionné.

En cas d'erreur, les valeurs de `errno` les plus fréquentes sont

errno	signification
EACCES	Le processus appelant n'a pas les droits suffisants pour obtenir l'attachement souhaité.
EINVAL	shmid a une valeur illégale.
...	...

Exemple 6.13 (shmat). Soit le programme suivant :

```
***** AttacheShm.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;
    char* pShm;

    cle = atoi(argv[1]);

    // Recuperation de l'identifiant de la memoire partagee
    if ((idShm = shmget(cle,0,0)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm);

    // Attachement a la memoire partagee
    if ((pShm = (char*)shmat(idShm,NULL,0)) == (char*)-1)
    {
        perror("Erreur de shmat");
        exit(1);
    }

    printf("pShm = %ld\n",pShm);

    // Utilisation de la memoire partagee
    if (argc >= 3) strcpy(pShm,argv[2]);    pas de shsnd, c'est via strcpy
    else printf("Contenu de la memoire partagee : --%s--\n",pShm);

    pause();
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./CreationShm 1234 100
idShm = 151027812
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x000004d2 151027812 student       600    100      0

# ./AttacheShm 1234 Hello &
[1] 17608
idShm = 151027812
pShm = 140459562266624
# ./AttacheShm 1234 &
[2] 17615
idShm = 151027812
pShm = 140296415887360
Contenu de la memoire partagee : --Hello--
# ./SuppressionShm 1234
idShm = 151027812
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x00000000 151027812 student       600    100      2      dest

# ps
PID   TTY      TIME      CMD
10969 pts/1    00:00:00 bash
17608 pts/1    00:00:00 AttacheShm
17615 pts/1    00:00:00 AttacheShm
17785 pts/1    00:00:00 ps
# kill -2 17608
[1]- Interrompre ./AttacheShm 1234 Hello
# kill -2 17615
[2]+ Interrompre ./AttacheShm 1234
# ipcs -m

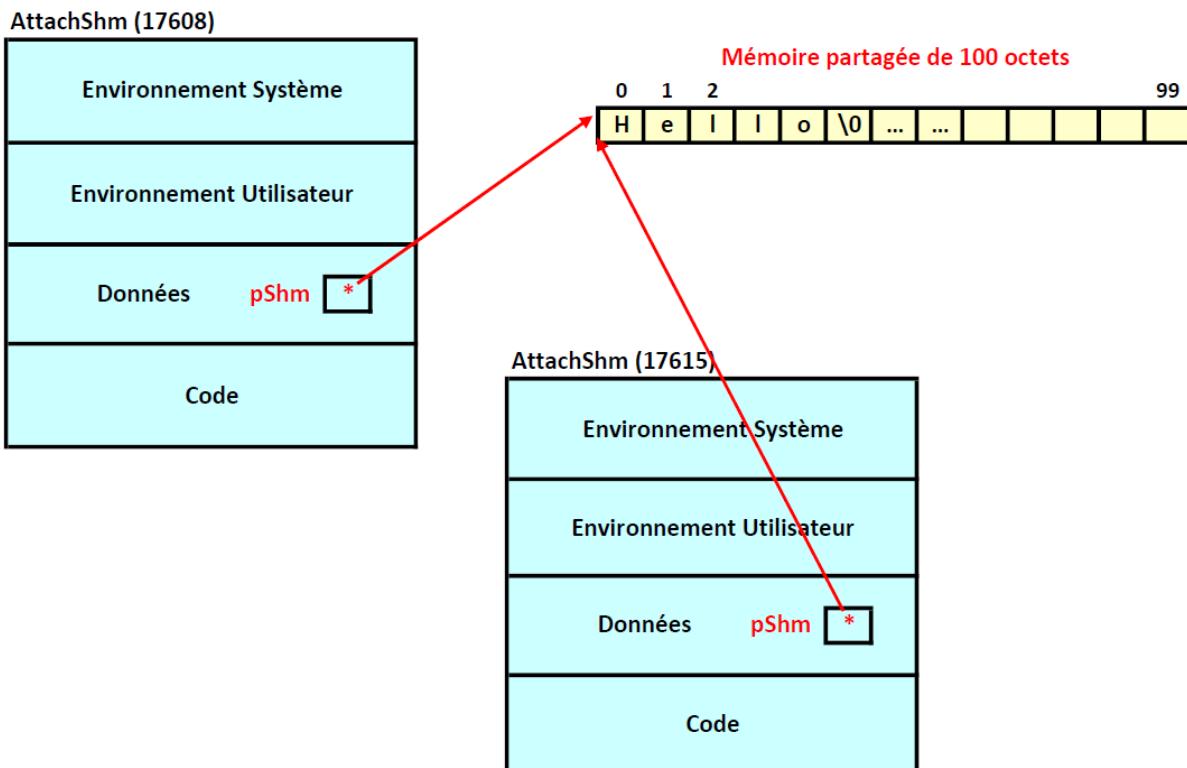
----- Segment de mémoire partagée -----
clef shmid propriétaire perms octets nattch états
#
```

On constate que :

1. Avant toute chose, une mémoire partagée de 100 octets a été créé grâce au programme **CreationShm** de l'**exemple 6.10**.
2. Ce programme récupère tout d'abord l'identifiant de la mémoire partagée dont la clé est précisée en premier argument de la ligne de commande. Il utilise cet identifiant pour obtenir un pointeur sur la mémoire partagée en utilisant la fonction **shmat**. Le **second paramètre** de **shmat** est **NULL** afin de laisser le noyau choisir la meilleure adresse d'attachement (**ce qu'il faut faire** tout le temps) tandis que le **troisième paramètre** est **0**

pour obtenir un attachement en **lecture/écriture**.

3. La **valeur de retour** de la fonction **shmat** est un pointeur que l'on **caste en (char*)** car on souhaite, dans cet exemple, utiliser la mémoire partagée comme une **simple chaîne de caractères**. Cette valeur de retour est le pointeur vers la mémoire partagée et est stockée dans la variable **pShm**. C'est via cette adresse que le processus peut écrire et lire dans la mémoire partagée.
4. En fonction du nombre d'arguments sur la ligne de commande, le programme **écrit** le 2ème argument de la ligne de commande (si **argc** est égal à **3**) **en mémoire partagée**, ou **affiche le contenu** de la mémoire partagée (si **argc** est égal à **2**). Dans les deux cas, la mémoire partagée s'utilise **comme une variable classique** du processus, ici à l'aide la fonction **strcpy** (pour l'**écriture en mémoire partagée**) ou l'aide la fonction **printf** (pour la **lecture en mémoire partagée**).
5. La **première exécution** écrit "Hello" en mémoire partagée tandis que la **seconde exécution** affiche le contenu de la mémoire partagée. Ceci est représenté schématiquement par



6. Bien que pointant vers la même mémoire partagée, les deux processus ont des **valeurs différentes** dans leur variable **pShm**. Il s'agit en fait d'une **adresse** dite **virtuelle**, associée par la fonction **shmat** à la même mémoire partagée. On dit encore que la mémoire partagée a été "**mappée**" à l'**espace adressable** du processus.
7. Après avoir manipulé la mémoire partagée, les deux processus **17608** et **17615** s'endorment sur l'appel système **pause**, afin de rester attachés à la mémoire partagée. Cela va permettre de mettre en évidence le fait que la **suppression effective de la mémoire partagée** ne se fera qu'après leur fin.
8. Le programme **SuppressionShm** de l'**exemple 6.12** est utilisé pour supprimer la mémoire partagée. Celle-ci n'est pas supprimée directement. il faut attendre que les processus qui y sont attachés soient tués par l'envoi manuel du signal **SIGINT** qui provoque leur terminaison directe.



6.4.5 Structurer une mémoire partagée

Une mémoire partagée peut contenir autre chose qu'une simple chaîne de caractères, il est possible d'y mettre ce que l'on veut : entiers, vecteurs, nombres réels, ... Mais dans ce cas, il est alors plus commode de structurer cette mémoire. Il suffit pour cela de **définir une structure** tout à fait classique et d'**alloquer une mémoire partagée de la taille de cette structure**. Enfin, lors de l'attachement à cette zone de mémoire partagée, le **pointeur** obtenu par la fonction shmat sera **casté en un pointeur dont le type correspond à cette structure**. Le principe est donc exactement le même qu'une allocation dynamique d'une structure par la fonction malloc.

Dans l'exemple qui suit,

- le programme **StructurerShm1** va créer une mémoire partagée structurée selon la structure MEM contenant :
 - un nombre réel (float)
 - une chaîne de caractères (char[20]),
 - un vecteur de 4 entiers (int[4]),
- le programme **StructurerShm1** va s'y attacher et y écrire des données dans les différents champs de la structure.
- le programme **StructurerShm2** va récupérer l'identifiant de cette mémoire partagée, s'y attacher et afficher ce qu'elle contient.

Exemple 6.14 (structurer une mémoire partagée). Soit le programme suivant :

```
***** StructurerShm1.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

typedef struct
{
    float val;
    char message[20];
    int vec[4];
} MEM;

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;
    MEM *p;

    cle = atoi(argv[1]);

    // Creation de la memoire partagee
    if ((idShm = shmget(cle,sizeof(MEM),IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
```

```

    perror("Erreur de shmget");
    exit(1);
}

printf("idShm = %d\n",idShm);

// Attachement a la memoire partagee
if ((p = (MEM*)shmat(idShm,NULL,0)) == (MEM*)-1)
{
    perror("Erreur de shmat");
    exit(1);
}

printf("p = %ld\n",p);

// Ecriture en memoire partagee
p->val = 3.75;
strcpy(p->message,"Texte dans la shm");
for (int i=1 ; i<=4 ; i++) p->vec[i-1] = i*10;

exit(0);
}

```

et le programme

```

/***** StructurerShm2.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

typedef struct
{
    float val;
    char message[20];
    int vec[4];
} MEM;

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;
    MEM *p;

    cle = atoi(argv[1]);

    // Recuperation de l'identifiant de la memoire partagee
    if ((idShm = shmget(cle,0,0)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm);

    // Attachement a la memoire partagee
    if ((p = (MEM*)shmat(idShm,NULL,SHM_RDONLY)) == (MEM*)-1)
    {
        perror("Erreur de shmat");
        exit(1);
    }
}

```

```

}

printf("p = %ld\n",p);

// Lecture en mémoire partagée
printf("val = %f\n",p->val);
printf("message = %s\n",p->message);
printf("vec = ");
for (int i=0 ; i<4 ; i++) printf("%d ",p->vec[i]);
printf("\n");

exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./StructurerShm1 1234
idShm = 159875096
p = 140268292997120
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid     propriétaire perms octets nattch états
0x000004d2 159875096 student       600    40      0

# ./StructurerShm2 1234
idShm = 159875096
p = 139621620400128
val = 3.750000
message = Texte dans la shm
vec = 10 20 30 40
#

```

On constate que :

1. La taille de la mémoire partagée est de **40** octets, ce qui correspond à 4 (float) + 20 (char) + 4 * 4 (int).
2. Le retour de la fonction **shmat** est **casté en MEM***, ce qui est nécessaire si on veut accéder facilement aux différents champs de la structure à l'aide de l'opérateur **->**
3. L'écriture et la lecture en mémoire partagée structurée se font par manipulation classique d'un pointeur de structure.



6.4.6 Détacher un processus d'une mémoire partagée : la fonction shmdt

Un processus peut à tout moment se détacher d'une mémoire partagée, c'est-à-dire que le **pointeur** qu'il possède vers cette mémoire **ne sera plus valide**, même si la mémoire partagée existe toujours. Remarquez qu'il peut à tout moment s'y attacher à nouveau en faisant un nouvel appel à la fonction **shmat**.

Pour se détacher d'une mémoire partagée, un processus peut utiliser la fonction :

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

int shmdt(const void *shmaddr);
```

où

- **shmaddr** est l'adresse d'attachement à la mémoire partagée que l'on désire rendre invalide.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EINVAL	shmaddr ne correspond pas à l'adresse d'une mémoire partagée.
...	...

Exemple 6.15 (shmdt). Soit le programme suivant :

```
***** TestShmdt.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc,char* argv[])
{
    int idShm;
    key_t cle;
    char *pShm1,*pShm2;

    cle = atoi(argv[1]);

    // Recuperation de l'identifiant de la memoire partagee
    if ((idShm = shmget(cle,0,0)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm);

    // Attachement a la memoire partagee
    if ((pShm1 = (char*)shmat(idShm,NULL,0)) == (char*)-1)
    { perror("Erreur de shmat (1)"); exit(1); }

    printf("pShm1 = %ld\n",pShm1);
```

```

// Attachement a la memoire partagee
if ((pShm2 = (char*)shmat(idShm,NULL,0)) == (char*)-1)
{ perror("Erreur de shmat (2)"); exit(1); }

printf("pShm2 = %ld\n",pShm2);

// Temporisation
sleep(5);

// Detachement de la memoire partagee
if (shmctl(pShm1, -1, IPC_RMID) == -1)
{
    perror("Erreur de shmctl");
    exit(1);
}

pause();

exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./CreationShm 1234 100
idShm = 161677336
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid      propriétaire perms octets nattch états
0x000004d2 161677336 student        600     100      0

# ./TestShmdt 1234 &
[1] 22261
idShm = 161677336
pShm1 = 140359946952704
pShm2 = 140359946948608
# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid      propriétaire perms octets nattch états
0x000004d2 161677336 student        600     100      2

# ipcs -m

----- Segment de mémoire partagée -----
clef      shmid      propriétaire perms octets nattch états
0x000004d2 161677336 student        600     100      1

#

```

On constate que :

1. Avant toute chose, une mémoire partagée de 100 octets a été créé grâce au programme **CreationShm** de l'**exemple 6.10**.
2. Le processus **s'attache à deux reprises** à la même mémoire partagée. Les adresses

pShm1 et **pShm2** sont différentes mais permettent toutes deux d'accéder à la même zone de mémoire.

3. Lors du **second appel à ipcs**, le processus est bloqué sur l'appel système **sleep** et on voit que **nattch = 2**. Cela signifie qu'il y a 2 pointeurs distincts qui pointent sur la mémoire partagée et non qu'il y a 2 processus distincts qui pointent vers cette mémoire partagée.
4. Après l'appel de la fonction **sleep**, le processus **se détache** de la mémoire partagée. Ceci est confirmé par le dernier appel à **ipcs**. L'adresse **pShm1** devient **invalid** mais la mémoire partagée existe toujours et le pointeur **pShm2** est **toujours valide**.



6.5 Les sémaphores

On se rend bien compte que l'utilisation simultanée d'une mémoire partagée par plusieurs processus différents peut poser problème. En effet, il ne faut pas qu'un processus en lise le contenu pendant qu'un autre processus est en train de la modifier, sous peine d'avoir une probable incohérence dans les données lues. Il faut donc réglementer l'accès à une mémoire partagée, voir même à toute ressource commune à plusieurs processus. Cette réglementation peut être obtenue à l'aide des **sémaphores** qui :

- permettent à plusieurs processus de **synchroniser leur exécution**. Les processus concernés vont ainsi pouvoir **attendre** qu'un autre processus ait terminé de modifier le contenu d'une ressource commune, avant, eux-mêmes, de pouvoir y accéder.
- peuvent, dans leur forme la plus simple, être vus comme des "**drapeaux**" qu'un processus lève ou descend afin de prévenir les autres processus qu'il utilise une ressource commune.
- peuvent être vus comme un système de "**feux rouges**" régulant les accès concurrents à une ressource commune.

Comme pour les files de messages et les mémoires partagées, une **table système** est dédiée à la gestion des sémaphores.

6.5.1 Principe général

Du point de vue programmation, un **sémaphore**

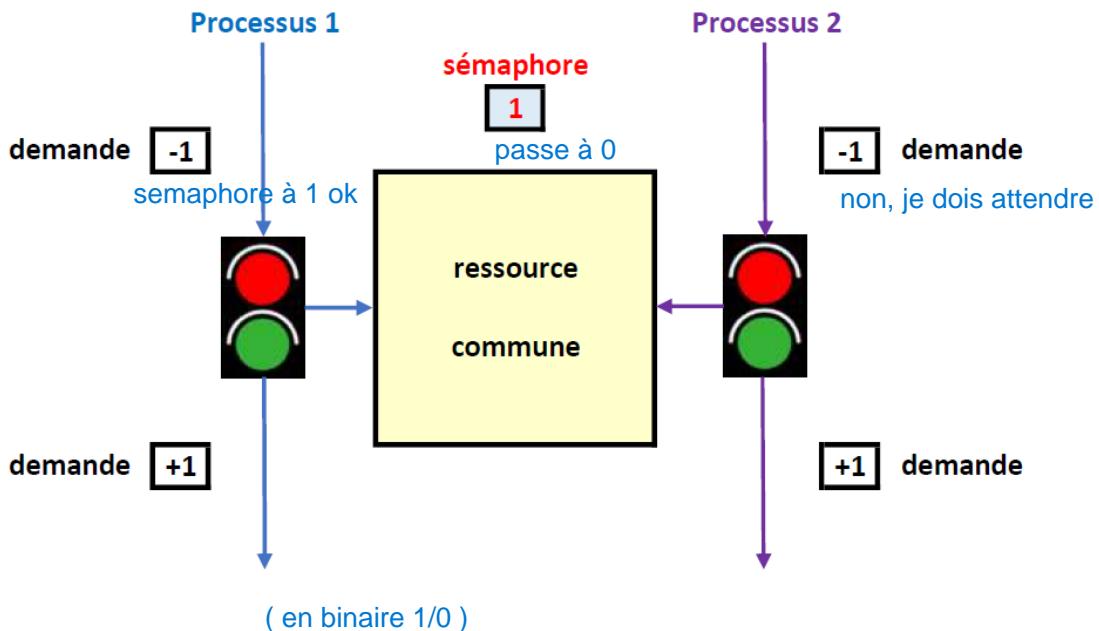
1. est un **compteur** ne pouvant contenir qu'une **valeur entière positive ou nulle**,
2. ne peut **jamais être négatif**.

Selon le contexte, plusieurs cas de figure peuvent se présenter :

- une seule ressource partagée,
- plusieurs ressources identiques partagées,
- plusieurs ressources différentes partagées,
- une synchronisation pure.

6.5.1.1 Un seule ressource partagée

Imaginons deux processus voulant utiliser une seule ressource commune. Ceci peut être schématisé par



Un seul **sémaphore** (correspondant à la seule ressource partagée) est utilisé et **initialisé à 1** avant toute utilisation (il agit ici en tant que "drapeau") :

- Une **valeur 1** correspond au fait que la **ressource est disponible (drapeau levé)**,
- Une **valeur 0** correspond au fait que la **ressource est en cours d'utilisation (drapeau baissé)**.

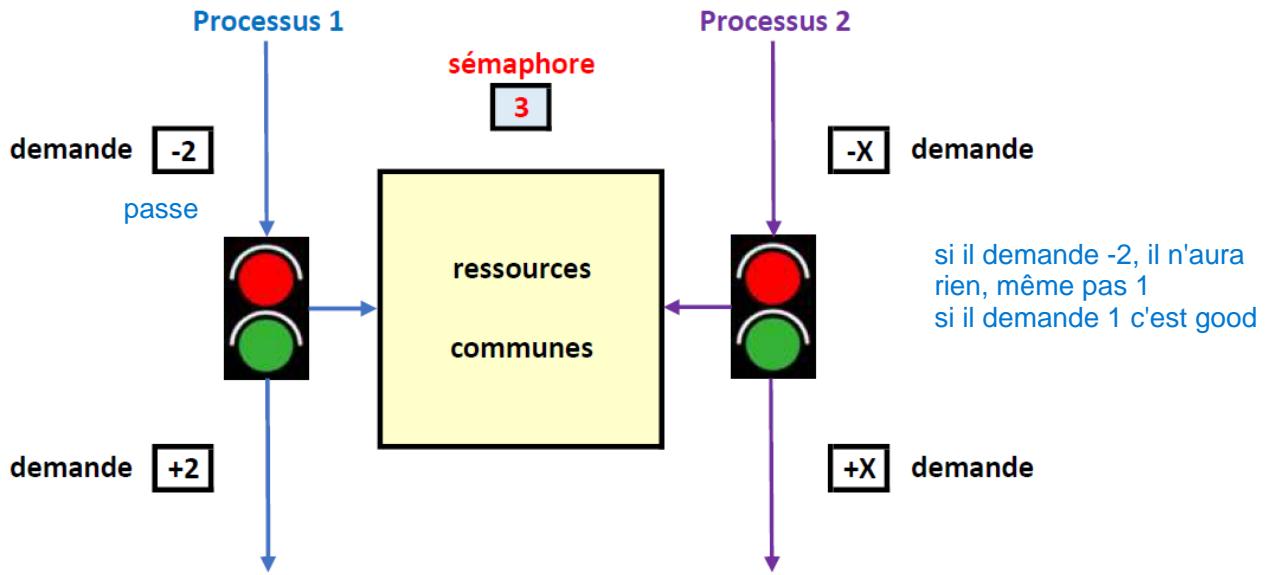
Imaginons la séquence exemple d'opérations suivante :

1. Le processus 1 atteind en premier le "feu rouge". Il souhaite décrémenter le sémaphore de 1 et arrive donc avec une **demande égale à -1**.
2. Le sémaphore étant à 1, celui-ci est disponible et le processus 1 peut le décrémenter. Le sémaphore passe donc à **0**. On dit encore que le processus 1 **prend possession du sémaphore** ("il baisse le drapeau") et il peut utiliser la ressource commune.
3. Le processus 2 atteind à son tour le "feu rouge". Il souhaite également décrémenter le sémaphore de 1 et a donc une **demande de -1**. Cependant, le sémaphore est à 0 et ne peut pas devenir négatif. En d'autres termes, "le drapeau est baissé" ou "le feu est rouge". Le processus 2 est donc **bloqué sur sa demande**.
4. Le processus 1 a terminé d'utiliser la ressource. Il va donc la **libérer le sémaphore**, "relever le drapeau", c'est-à-dire incrémenter celui-ci de 1, il a donc une **demande de +1** (demande qui n'est **jamaïs bloquante** car aucun risque que le sémaphore ne devienne négatif). Le sémaphore repasse donc à **1**.
5. La demande du processus 2 est alors acceptée, le sémaphore passe à **0** et le processus 2 utilise à son tour la ressource.
6. Après avoir utilisé la ressource, le processus 2 incrémente le sémaphore de 1 (**demande de +1**).
7. Le sémaphore a alors **retrouvé sa valeur initiale de 1** et peut à nouveau être utilisé.

Il s'agit de l'utilisation la plus simple et la plus classique d'un sémaphore qui porte alors également le nom de “**mutex**” (pour **mutuellement exclusif** - quand un processus utilise la ressource, les autres en sont exclus).

6.5.1.2 Plusieurs ressources identiques partagées

Imaginons deux processus voulant utiliser plusieurs ressources identiques partagées. Ceci peut être schématisé par



Un exemple concret pourrait être deux ouvriers (les deux processus) qui partagent la même boîte à outils (les ressources partagées) qui contient 3 tournevis (les ressources identiques).

Un seul sémaphore (correspondant au seul type de ressource partagée - les “tournevis”) est utilisé et **initialisé à 3** (3 car la boîte à outils contient 3 tournevis) avant toute utilisation (le sémaphore n'est plus vu comme un drapeau mais plutôt comme un compteur de ressources disponibles) :

- Une **valeur positive** correspond au fait qu'un certain nombre de ressources sont disponibles,
- Une **valeur 0** correspond au fait qu'il n'y a aucune ressource disponible.

Imaginons la séquence exemple d'opérations suivante :

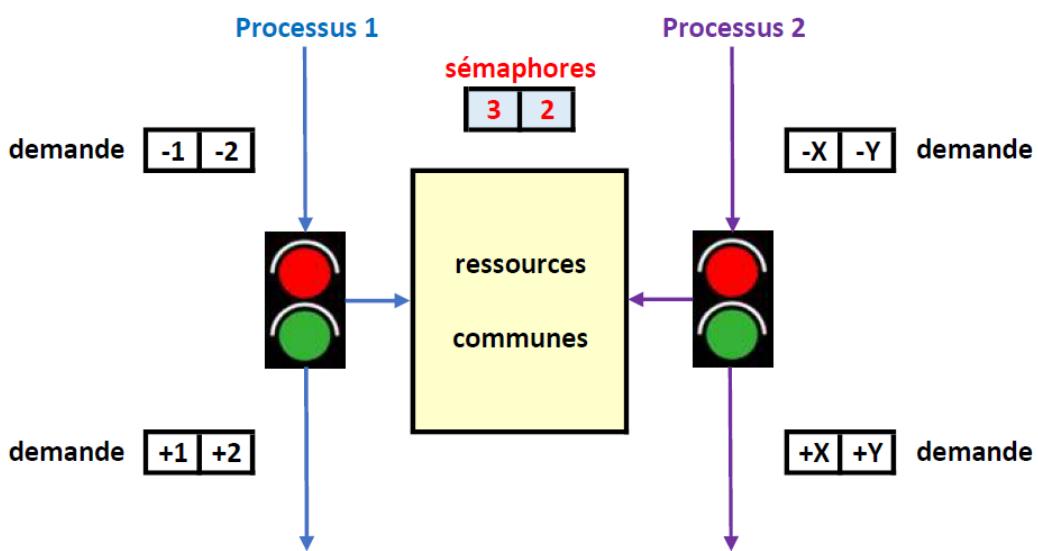
1. Le processus 1 atteind en premier le “feu rouge”. Il souhaite utiliser 2 tournevis, donc décrémenter le sémaphore de 2 et arrive avec une **demande égale à -2**.
2. Le sémaphore étant à 3, il y a assez de tournevis disponibles ($3 - 2 = 1 \geq 0$) et le processus 1 peut le décrémenter de 2. Le sémaphore passe donc à **1**. Le processus 1 continue son exécution et utilise deux tournevis de la boîte à outils.
3. Le processus 2 atteind à son tour le “feu rouge”. Il souhaite utiliser X tournevis, donc décrémenter le sémaphore de X et a donc une **demande de $-X$** . Selon la valeur de X (que nous limiterons à 1 ou 2 pour l'exemple), la suite des événements est différente :
 - (a) Si $X = 2$, il n'y a plus assez de tournevis ($1 - 2 = -1 < 0$). Bien qu'il reste un tournevis, le processus 2 ne le prend pas (c'est un **mécanisme de “tout ou rien”**) et reste donc **bloqué sur sa demande**.

- (b) Si $X = 1$, la boîte à outils contient assez de tournevis ($1 - 1 \geq 0$). La demande du processus 2 est acceptée et le sémaphore est décrémenté de 1, il passe donc à **0**. Le processus 2 peut donc continuer son exécution et utiliser le tournevis.
4. Le processus 1 a terminé d'utiliser ses 2 tournevis. Il va donc incrémenter le sémaphore de 2, il a donc une **demande de +2** (demande qui n'est **jamais bloquante** car aucun risque que le sémaphore ne devienne négatif). Le sémaphore repasse donc à $3 = 1 + 2$ (si $X = 2$) ou à $2 = 0 + 2$ (si $X = 1$)
 5. Si $X = 2$, le processus 2 était bloqué et sa demande est à présent acceptée, le sémaphore passe à **$1 = 3 - 2$** et le processus 2 utilise à son tour 2 tournevis.
 6. Après avoir utilisé les ressources, le processus 2 incrémente le sémaphore de X (**demande de +X**).
 7. Le sémaphore a alors **retrouvé sa valeur initiale de 3** et peut à nouveau être utilisé.

En terme de programmation, on dit que l'opération de demande est une **opération atomique**, c'est-à-dire que la demande est **acceptée totalement ou pas du tout**.

6.5.1.3 Plusieurs ressources différentes partagées

Imaginons deux processus voulant utiliser plusieurs ressources différentes partagées. Ceci peut être schématisé par



Un exemple concret pourrait être deux ouvriers (les deux processus) qui partagent la même boîte à outils (les ressources partagées) qui contient 3 tournevis et 2 marteaux (les ressources différentes).

Deux sémaphores (correspondant aux deux types de ressource partagée - les "tournevis" et les "marteaux") sont utilisés et **initialisés respectivement à 3 et 2** (car la boîte à outils contient 3 tournevis et 2 marteaux) avant toute utilisation (le sémaphore n'est à nouveau plus vu comme un drapeau mais plutôt comme un compteur de ressources disponibles) :

Imaginons la séquence exemple d'opérations suivante :

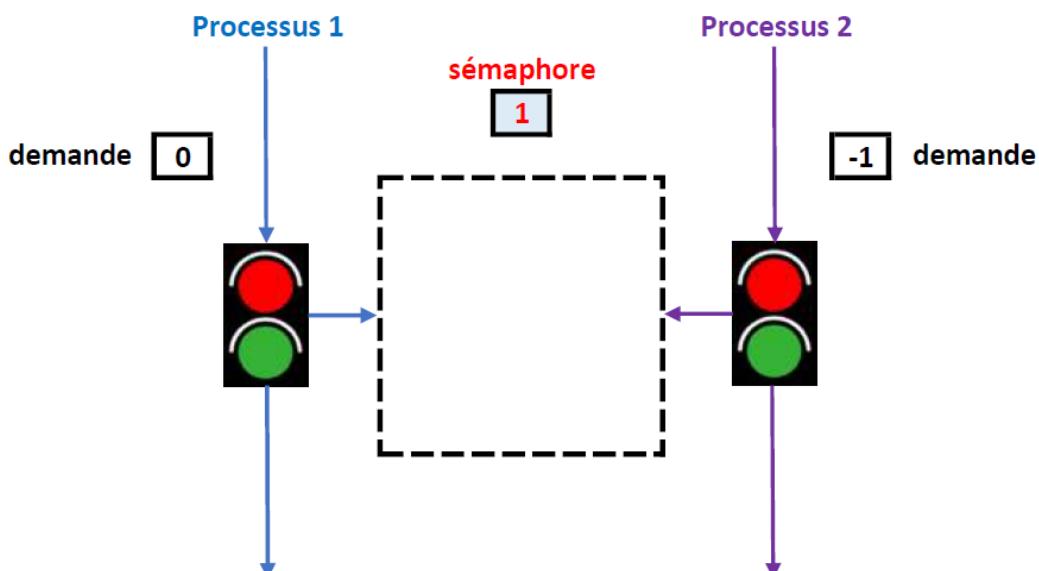
1. Le processus 1 atteind en premier le "feu rouge". Il souhaite utiliser 1 tournevis et 2 marteaux, donc décrémenter le 1er sémaphore de 1 et le 2ème sémaphore de 2. Il arrive donc avec une **demande de [-1][-2]**.

2. Il y a assez de tournevis ($3 - 1 = 2 \geq 0$) et de marteaux ($2 - 2 = 0 \geq 0$) disponibles et la demande du processus 1 est acceptée. Les sémaphores passent donc à **2** et **0**. Le processus 1 peut continuer son exécution et utiliser un tournevis et deux marteaux de la boîte à outils.
3. Le processus 2 atteind à son tour le "feu rouge". Il souhaite utiliser X tournevis et Y marteaux, donc décrémenter le 1er sémaphore de X et le 2ème sémaphore de Y . Il a donc une **demande de $[-X][-Y]$** . Sa demande ne sera acceptée que si il reste assez de tournevis **et** de marteaux, c'est-à-dire
 - (a) si $2 - X \geq 0$, **ET**
 - (b) si $0 - Y \geq 0$, c'est-à-dire si $Y = 0$
Dans ce cas, les sémaphores sont décrémentés en conséquence. Sinon, le processus 2 reste bloqué sur sa demande.
4. Le processus 1 a terminé d'utiliser son tournevis et ses deux marteaux. Il va donc incrémenter le 1er sémaphore de 1 et le 2ème sémaphore de 2, il a donc une **demande de $[+1][+2]$** (demande qui n'est **jamais bloquante** car aucun risque que les sémaphores ne deviennent négatifs). Les deux sémaphores sont alors incrémentés en conséquence.
5. Après avoir accéder aux ressources, le processus 2 incrémente le 1er sémaphore de X et le 2ème sémaphore (**demande de $[+X][+Y]$**).
6. Les sémaphores ont alors **retrouvé leurs valeurs initiales de $[3][2]$** et peuvent à nouveau être utilisés.

Même s'il y a assez de tournevis (marteaux), la demande d'un ouvrier ne sera acceptée que si il y a assez de marteaux (tournevis) également. Comme déjà mentionné, il s'agit d'une **opération atomique**, c'est-à-dire que la demande est **acceptée totalement ou pas du tout**.

6.5.1.4 Synchronisation pure

Imaginons un ou plusieurs processus (dans l'exemple qui suit, le processus 1 uniquement) voulant **synchroniser son (leur) exécution sur un événement** correspondant au **passage à 0 d'un compteur** (cela peut être un compteur de processus, de tâches à réaliser, etc...). Ce compteur sera décrémenté par un ou plusieurs autres processus (dans l'exemple qui suit, le processus 2 uniquement). Ceci peut être schématisé par



Un seul sémaphore (correspondant au compteur en question) est utilisé et doit présenter une valeur positive à un moment donné.

Imaginons la séquence exemple d'opérations suivante :

1. Le processus 1 atteind en premier le "feu rouge". Il souhaite attendre que le sémaphore passe à 0. Pour cela, il a une demande égale à 0 et il reste bloqué sur sa demande tant que le sémaphore est positif.
2. Le processus 2 atteind à son tour le "feu rouge". Il décrémente le sémaphore de 1 (pour une raison qui lui est propre et qui dépend du sens donné au compteur) et a donc une demande de -1. Le sémaphore étant à 1, celui-ci peut être décrémenté et passe donc à 0. Le processus 2 continue alors son exécution. Cette décrémentation a pour effet de notifier le processus 1.
3. Le processus 1 est réveillé et peut à présent évaluer la valeur du sémaphore. Si celui-ci est à 0 (ce qui est le cas ici), il continue son exécution. Sinon, il resterait bloqué jusqu'à la prochaine notification.

Ici, il n'y pas vraiment de ressource commune dont l'accès doit être protégé (d'où les pointillés sur le schéma). Cette technique permet surtout de synchroniser plusieurs processus afin qu'il reprennent leur exécution en même temps lorsqu'un événement s'est produit (ici l'annulation du sémaphore).

6.5.1.5 Premier aperçu des fonctions associées aux sémaphores

En regard de ce qui a été décrit ci-dessus, plusieurs tâches relatives aux sémaphores doivent être envisagées :

- la création d'un ensemble de sémaphores (constitué d'un ou plusieurs sémaphores) : pour cela, il sera nécessaire d'utiliser la fonction `semget`.
- l'initialisation d'un ou plusieurs sémaphores avant leur utilisation : pour cela, il sera nécessaire d'utiliser la fonction `semctl`.
- les demandes de synchronisation, c'est-à-dire les demandes d'incrémentation ou de décrémentation de sémaphores : pour cela, il sera nécessaire d'utiliser la fonction `semop`.

Ces fonctions sont décrites en détails ci-dessous.

6.5.2 Structure interne d'un ensemble de sémaphores

La table système des sémaphores comprend une entrée distincte pour chaque ensemble de sémaphores (qui peut donc comporter plusieurs sémaphores) et chaque entrée est structurée par la structure suivante :

```

struct semid_ds
{
    struct ipc_perm sem_perm; /* operation permission struct */
    time_t           sem_otime; /* last semop() time */
    time_t           sem_ctime; /* last time changed by semctl() */
    unsigned int     sem_nsems; /* number of semaphores in set */

    struct sem      *sem_base;
};

```

où

- **sem_perm** est la structure commune à tous les IPC, reprenant la clé, le créateur, le propriétaire, les droits, ...
- **sem_otime** est la date/heure de la dernière opération de synchronisation réalisée par un appel à la fonction semop (voir plus loin).
- **sem_ctime** est la date/heure de création et/ou dernière modification de paramètres de l'ensemble de sémaphores.
- **sem_nsems** est le **nombre de sémaphores de l'ensemble**.
- **sem_base** est un pointeur vers le premier élément du vecteur contenant les sémaphores individuels faisant partie de l'ensemble.

La **structure sem** représente un sémaphore en mémoire et est définie par

```

struct sem
{
    unsigned short  semval;
    pid_t          sempid;
    unsigned short  semncnt;  combien sont bloqué au feu rouge
    unsigned short  semzcnt;  combien attende que le semaphore tombe à 0
};

```

où

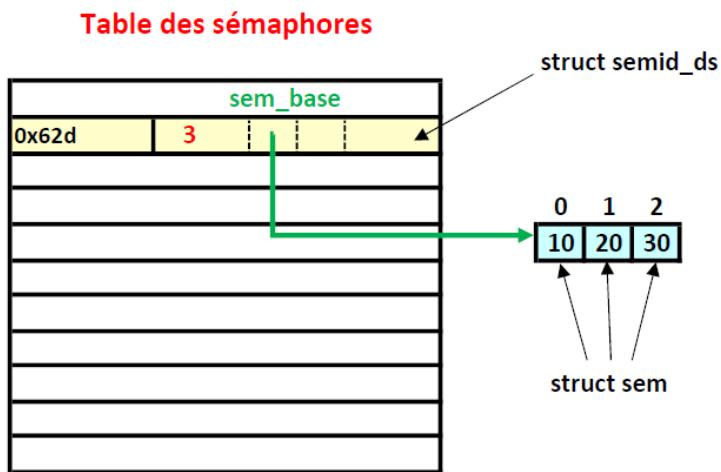
- **semval** est la **valeur actuelle du sémaphore**,
- **sempid** est le PID du dernier processus ayant réalisé une opération de synchronisation sur ce sémaphore à l'aide de la fonction semop,
- **semncnt** est le nombre de processus qui attendent actuellement que le sémaphore reprennent une valeur positive,
- **semzcnt** est le nombre de processus qui attendent actuellement que le sémaphore passe à 0.

Faisant partie de la famille des IPC au même titre que les files de messages et les mémoires partagées, un ensemble de sémaphores est caractérisé par

- une **clé** commune partagée et,

- un **identifiant** qui permet à chaque processus disposant de la clé de se **synchroniser** à l'aide de cet ensemble de sémaphores.

Le schéma ci-dessous illustre la structure interne d'un ensemble de 3 sémaphores (de valeurs 10, 20 et 30), de clé 0x62d :



Chaque sémaphore d'un ensemble de sémaphore est **identifié par son indice** dans le vecteur pointé par **`sem_base`**.

6.5.3 Obtention d'un ensemble de sémaphores : la fonction `semget`

L'obtention d'un ensemble de sémaphores se fait à l'aide de l'appel système **`semget`** qui permet, selon le cas, de

1. **créer** un ensemble de sémaphores et d'**obtenir son identifiant**,
2. **obtenir l'identifiant** d'un ensemble de sémaphores qui **existe déjà**.

C'est donc la même fonction qui permet à un processus de créer une nouvel ensemble de sémaphores ou de récupérer l'identifiant d'un ensemble de sémaphores qui a été créé par un autre processus.

Le prototype de cette fonction est :

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
  
```

où

- **key** est la clé de l'ensemble de sémaphores que l'on désire créer, ou de l'ensemble de sémaphores existant dont on veut récupérer l'identifiant,

- **nsems** est le **nombre de sémaphores de l'ensemble** que l'on désire créer.
- **semflg** est un paramètre entier permettant de préciser si l'on souhaite créer une nouvel ensemble de sémaphores ou simplement récupérer l'identifiant d'un ensemble de sémaphores existant.

Plusieurs combinaisons de paramètres sont possibles :

key	nsems	semflg	action
IPC_PRIVATE	> 0		Création d'un ensemble de sémaphores dont on laisse le système choisir la clé. Cela est suffisant lorsque l'ensemble de sémaphores est utilisé uniquement par des processus parents (qui héritent de l'identifiant de l'ensemble de sémaphores sans devoir faire un nouvel appel à la fonction semget).
> 0	> 0	IPC_CREAT droits	Création d'un ensemble de nsems sémaphores dont la clé est précisée par key . Les droits correspondent aux permissions de lecture/écriture/exécution pour le propriétaire, le groupe et les autres (exactement comme cela est fait pour les fichiers avec la fonction open).
> 0	= 0	= 0	Récupération (sans création) de l'identifiant de l'ensemble de sémaphores dont la clé est key .

On peut également combiner (par un OU bit à bit) **IPC_CREAT** et **IPC_EXCL**. Dans ce cas, si l'on souhaite créer un ensemble de sémaphores dont la **clé est déjà utilisée** par un autre ensemble de sémaphores existant, cela provoquera une **erreur** au lieu d'**écraser** l'ensemble de sémaphores existant. Ceci est similaire au comportement de la fonction open avec la combinaison **O_CREAT | O_EXCL**.

Lors de la **création** de l'ensemble de sémaphores (avec **key > 0**) :

- le **créateur** et **propriétaire** (structure ipc_perm) sont initialisés avec le UID du processus appelant. Il en est de même pour le groupe propriétaire et le groupe créateur.
- le champ **mode** de la structure ipc_perm est initialisé avec les droits demandés.
- Le champs **sem_nsems** de la structure semid_ds prend la valeur **nsems**.
- Le champ **sem_ctime** de la structure semid_ds est rempli avec l'heure actuelle.

Il est important de noter qu'une fois créés, **les sémaphores ont une valeur arbitraire**. il est **nécessaire de les initialiser** correctement en utilisant la fonction **semctl** (voir plus loin).

Valeurs de retour

Cette fonction retourne

- l'**identifiant de l'ensemble de sémaphores** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EEXIST	Un ensemble de sémaphores existe déjà pour cette clé key et IPC_CREAT IPC_EXCL est spécifié.
EACCES	Un ensemble de sémaphores existe déjà pour cette clé key, mais le processus appelant n'a pas les permissions pour y accéder.
ENOENT	Aucun ensemble de sémaphores n'existe associé à la clé key et semflg ne contient pas IPC_CREAT.
ENOSPC	Le nombre maximal de sémaphores ou d'ensembles de sémaphores sur le système est atteint.
...	...

Exemple 6.16 (semget pour une création). Soit le programme suivant :

```
***** CreationSem.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc,char* argv[])
{
    int idSem;
    key_t cle;
    int nb;

    cle = atoi(argv[1]);
    nb = atoi(argv[2]);

    if ((idSem = semget(cle,nb,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        perror("Erreur de semget");
        exit(1);
    }

    printf("idSem = %d\n",idSem);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -s

----- Tableaux de sémaphores -----
clef semid propriétaire perms nsems

# ./CreationSem 1234 3
idSem = 131072
# ipcs -s

----- Tableaux de sémaphores -----
clef      semid  propriétaire perms nsems
0x000004d2 131072 student      600    3

# ./CreationSem 1234 5
Erreur de semget: File exists
# ipcs -s -i 131072
Tableaux de sémaphores semid=131072
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Non initialisé
ctime = Thu Aug 5 10:14:54 2021
semnum  valeur  ncount  zcount  PID
0        0        0        0        0
1        0        0        0        0
2        0        0        0        0
#
```

On constate que :

1. Ce programme permet de **créer un ensemble de sémaphores** dont la clé et le nombre de sémaphores sont précisés en **premier et second argument** de la ligne de commande.
2. Lors de la première exécution, l'ensemble de sémaphores est créé correctement. La **clé 0x4d2** apparaissant lors de l'appel de la commande **ipcs** correspond à la **valeur hexadécimale de 1234**.
3. L'**identifiant 131072** a été attribué à l'ensemble de sémaphores lors de l'appel système **semget** et stocké dans la variable **idSem**.
4. Les **droits** associés à l'ensemble de sémaphores sont **600**, c'est-à-dire un accès en **lecture/écriture pour le propriétaire** et rien pour les autres. Les processus dont le UID est le propriétaire de l'ensemble de sémaphores pourront l'utiliser pour synchroniser leur exécution.
5. Lors de la seconde exécution, on a voulu créer un second ensemble de sémaphores avec la **même clé 1234**, ce qui a provoqué une **erreur** vu que cette clé est déjà utilisée par un ensemble de sémaphores existant et que **IPC_CREAT | IPC_EXCL** a été spécifié dans l'appel de **semget**.
6. La commande **ipcs** (avec ou sans l'option **-i**) a permis d'afficher les informations de l'ensemble de sémaphores. On y voit par exemple le **nombre de sémaphores de l'ensemble (3)**.
7. L'utilisation de la commande **ipcs** avec l'option **-i** permet en plus de **visualiser les valeurs de sémaphores** de l'ensemble dont l'identifiant est passé en paramètre. On

observe que les 3 sémaphores de l'ensemble ont été initialisés à 0. De plus, on dispose des informations de la **structure sem** pour chaque sémaphore de l'ensemble.



Exemple 6.17 (semget pour une récupération). Soit le programme suivant :

```
***** RecupSem.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc,char* argv[])
{
    int idSem;
    key_t cle;

    cle = atoi(argv[1]);

    if ((idSem = semget(cle,0,0)) == -1)
    {
        perror("Erreur de semget");
        exit(1);
    }

    printf("idSem = %d\n",idSem);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -s

----- Tableaux de sémaphores -----
clef      semid  propriétaire perms nsems
0x000004d2 131072 student      600     3

# ./RecupSem 1234
idSem = 131072
# ./RecupSem 111
Erreur de semget: No such file or directory
#
```

On constate que :

1. Ce programme permet de **récupérer l'identifiant** d'un ensemble de sémaphores dont la clé est précisée en **premier argument** de la ligne de commande.
2. La première exécution a permis de récupérer l'**identifiant 131072** de l'ensemble de sémaphores existant de clé **1234**.
3. La seconde exécution a provoqué une **erreur** car on a tenté de récupérer l'identifiant de l'ensemble de sémaphores de **clé 111** qui n'existe pas.



6.5.4 Initialisation/suppression d'un ensemble de sémaphores : semctl

Comme déjà mentionné, avant toute utilisation, il est **nécessaire d'initialiser correctement les sémaphores** d'un ensemble qui vient d'être créé, et cela en fonction de leur utilisation souhaitée (voir "Principe général" des sémaphores ci-dessus).

L'appel système qui permet d'**initialiser** un sémaphore/ensemble de sémaphore est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

où

- **semid** est l'**identifiant** de l'ensemble de sémaphores que l'on désire manipuler,
- **semnum** est le **numéro du sémaphore** (l'**indice** dans le vecteur pointé par **sem_base**) sur lequel on veut agir.
- **cmd** est un paramètre entier permettant de spécifier la **commande** (c'est-à-dire l'**action**) à réaliser sur l'ensemble de sémaphores,
- **...** est un paramètre optionnel décrit ci-dessous.

La fonction **semctl** peut donc s'utiliser avec 3 ou 4 paramètres selon la valeur de cmd. Quand il y en a quatre, le quatrième est de type **union semun**. Le programme appelant **doit définir cette union** de la façon suivante :

```
union semun
{
    int             val;      /* Valeur pour SETVAL */
    struct semid_ds *buf;    /* Tampon pour IPC_STAT, IPC_SET */
    unsigned short  *array;   /* Tableau pour GETALL, SETALL */
};
```

où

- **val** permet d'initialiser un seul sémaphore de l'ensemble avec **cmd = SETVAL**,
- **buf** est l'adresse d'une structure **semid_ds** à utiliser avec **cmd = IPC_STAT** ou **cmd = IPC_SET**,
- **array** est l'adresse d'un tableau d'entiers à utiliser avec **cmd = GETALL** ou **cmd = SETALL**.

Si on appelle **arg** la variable de type **union semun**, voici quelques commandes/actions possibles :

cmd	semnum	arg	action
IPC_RMID	0	/	Suppresion de l'ensemble de sémaphores d'identifiant semid .
IPC_STAT	0	arg.buf	Les informations de l'entrée correspondant à l'identifiant semid de la table des sémaphores sont récupérées en local dans la structure semid_ds pointée par arg.buf.
IPC_SET	0	arg.buf	Les informations de l'entrée correspondant à l'identifiant semid de la table des sémaphores sont modifiées par celles contenues dans la structure semid_ds pointée par arg.buf. Seuls les champs suivants de la structure peuvent être mis à jour : sem_perm.uid, sem_perm.gid et les 9 bits de poids faible de sem_perm.mode.
SETVAL	≥ 0	arg.val	Initialisation du semnum ième sémaphore de l'ensemble avec la valeur arg.val .
SETALL	0	arg.array	Initialisation de tous les sémaphores de l'ensemble avec les valeurs contenues dans arg.array .
GETVAL	≥ 0	/	Retour de la valeur du semnum ième sémaphore de l'ensemble.
GETALL	0	arg.array	Copie dans la variable locale arg.array de la valeur semval de chaque sémaphore de l'ensemble.
GETPID	≥ 0	/	Retour du PID du dernier processus qui a manipulé le semnum ième sémaphore de l'ensemble.
GETNCNT	≥ 0	/	Retour du nombre de processus qui attendent que le semnum ième sémaphore redevienne positif.
GETZCNT	≥ 0	/	Retour du nombre de processus qui attendent que le semnum ième sémaphore passe à 0.

Comme on peut le remarquer, la fonction **semctl** est une véritable boîte à outils, elle permet

- de **supprimer** un ensemble de sémaphores,
- d'**initialiser** un ou tous les sémaphores d'un ensemble,
- de récupérer diverses **informations** sur chaque sémaphore de l'ensemble (les champs de la structure sem de chaque sémaphore),
- un “**contrôle**” de l'ensemble de sémaphores, comme par exemple en modifier le propriétaire ou encore les droits d'accès.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès (pour cmd égal à IPC_RMID, IPC_STAT, IPC_SET, SETVAL, SETALL, GETALL),
- la **valeur du semnum ième sémaphore** si cmd = GETVAL,
- la **valeur du PID du dernier processus** qui a manipulé le semnum ième sémaphore si cmd = GETPID,
- le **nombre de processus** qui attendent que le semnum ième sémaphore redevienne positif si cmd = GETNCNT,
- le **nombre de processus** qui attendent que le semnum ième sémaphore passe à 0 si cmd = GETZCNT,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	L'argument cmd vaut IPC_STAT, GETALL, GETPID, GETNCNT, GETZCNT, SETALL ou SETVAL mais le processus appelant n'a pas les droits d'accès suffisant sur l'ensemble de sémaphores d'identifiant semid .
EINVAL	cmd ou semid ont une valeur illégale.
EFAULT	arg.buf ou arg.array pointe en dehors de l'espace d'adressage autorisé.
EPERM	L'argument cmd vaut IPC_SET ou IPC_RMID mais l'UID effectif du processus appelant n'est pas le créateur (comme indiqué dans sem_perm.cuid) ou le propriétaire (comme indiqué dans sem_perm.uid) de l'ensemble de sémaphores.
ERANGE	L'argument cmd vaut SETALL ou SETVAL mais la valeur de semval (pour l'ensemble ou pour certains sémaphores) est inférieure à 0 ou supérieure à la valeur SEMVMX.
...	...

Exemple 6.18 (semctl pour une initialisation). Soit le programme suivant :

```
***** InitSem.cpp ****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;

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

```
{  
    int idSem;  
    key_t cle;  
  
    cle = atoi(argv[1]);  
  
    // Recuperation de l'ensemble de semaphores  
    if ((idSem = semget(cle,0,0)) == -1)  
    {  
        perror("Erreur de semget");  
        exit(1);  
    }  
  
    printf("idSem = %d\n",idSem);  
  
    // Initialisation de tous les semaphores  
    unsigned short v[] = {10, 5, 30};  
    arg.array = v;  
  
    if (semctl(idSem,0,SETALL,arg) == -1)  
    {  
        perror("Erreur de semctl (1)");  
        exit(1);  
    }  
  
    // Temporisation  
    sleep(10);  
  
    // Initialisation d'un seul semaphore  
    arg.val = 20;  
  
    if (semctl(idSem,1,SETVAL,arg) == -1)  
    {  
        perror("Erreur de semctl (2)");  
        exit(1);  
    }  
  
    exit(0);  
}
```

dont un exemple d'exécution fournit

```
# ipcs -s -i 131072

Tableaux de sémaphores semid=131072
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Non initialisé
ctime = Thu Aug 5 10:14:54 2021
semnum  valeur  ncount  zcount  PID
0        0        0        0        0
1        0        0        0        0
2        0        0        0        0

# ./InitSem 1234 &
[2] 2817
idSem = 131072
# ipcs -s -i 131072

Tableaux de sémaphores semid=131072
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Non initialisé
ctime = Thu Aug 5 17:15:50 2021
semnum  valeur  ncount  zcount  PID
0        10       0        0        2817
1        5        0        0        2817
2        30       0        0        2817

# ipcs -s -i 131072

Tableaux de sémaphores semid=131072
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Non initialisé
ctime = Thu Aug 5 17:16:00 2021
semnum  valeur  ncount  zcount  PID
0        10       0        0        2817
1        20       0        0        2817
2        30       0        0        2817

[2]+ Fini ./InitSem 1234
#
```

On constate que :

1. Avant exécution du programme, tous les sémaphores de l'ensemble sont à **0**.
2. Ce programme récupère tout d'abord l'identifiant de l'ensemble de sémaphores dont la **clé** est précisée en **premier argument** de la ligne de commande.
3. L'**union semun** a dû être **définie explicitement** dans le programme et la variable **arg** a été déclarée dans la foulée.

4. Le **premier appel** à **semctl initialise l'ensemble des sémaphores** avec les valeurs [10][5][30] avec cmd = **SETALL** en utilisant **arg.array** qui a été affecté à l'adresse du vecteur **v** contenant les valeurs d'initialisation. Ceci est confirmé par le second appel à la commande **ipcs**.
5. Le **second appel** à **semctl initialise** uniquement **le sémaphore no 1** avec la valeur **10** avec cmd = **SETPVAL** en utilisant **arg.val**. Ceci est confirmé par le dernier appel à la commande **ipcs**.



Exemple 6.19 (semctl pour une visualisation/suppresion). Soit le programme suivant :

```
***** SuppressionSem.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/IPC.h>
#include <sys/sem.h>

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;

int main(int argc,char* argv[])
{
    int idSem;
    key_t cle;
    int ret;

    cle = atoi(argv[1]);

    // Recuperation de l'ensemble de semaphores
    if ((idSem = semget(cle,0,0)) == -1)
    {
        perror("Erreur de semget");
        exit(1);
    }

    printf("idSem = %d\n",idSem);

    // Recuperation des valeurs de tous les semaphores
    unsigned short v[3];
    arg.array = v;

    if (semctl(idSem,0,GETALL,arg) == -1)
    {
        perror("Erreur de semctl (1)");
        exit(1);
    }

    printf("Valeurs des semaphores : ");
    for(int i=0 ; i<3 ; i++) printf("%d ",v[i]);
    printf("\n");

    // Recuperation du semaphore no 2
    if ((ret = semctl(idSem,2,GETVAL)) == -1)
```

```

{
    perror("Erreur de semctl (2)");
    exit(1);
}

printf("Valeur du semaphore no 2 : %d\n",ret);

// Suppression de l'ensemble de semaphores
if (semctl(idSem,0,IPC_RMID) == -1)
{
    perror("Erreur de semctl (3)");
    exit(1);
}

exit(0);
}

```

dont un exemple d'exécution fournit

```

# ipcs -s -i 131072

Tableaux de sémaphores semid=131072
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Non initialisé
ctime = Thu Aug 5 17:16:00 2021
semnum  valeur  ncount  zcount  PID
0        10      0        0        2817
1        20      0        0        2817
2        30      0        0        2817

# ./SuppressionSem 1234
idSem = 131072
Valeurs des semaphores : 10 20 30
Valeur du semaphore no 2 : 30
# ipcs -s

----- Tableaux de sémaphores -----
clef semid propriétaire perms nsems

#

```

On constate que :

1. Avant exécution du programme, les sémaphores de l'ensemble ont les valeurs **[10][20][30]**.
2. Ce programme récupère tout d'abord l'identifiant de l'ensemble de sémaphores dont la **clé** est précisée en **premier argument** de la ligne de commande.
3. L'**union semun** est à nouveau **définie explicitement** dans le programme et la variable **arg** est déclarée dans la foulée.
4. Le **premier appel** à **semctl** récupère en local, dans la variable **v**, les valeurs de l'ensemble des sémaphores avec cmd = **GETALL**.
5. Le **second appel** à **semctl** récupère en local, en retour de la fonction, dans la variable **ret**, la valeur du **sémaphore no 2** avec cmd = **GETVAL**.

6. Le **dernier appel à semctl supprime l'ensemble de sémaphores** avec cmd = **IPC_RMID**. Ceci est confirmé par le dernier appel de la commande **ipcs**.



Remarque importante

Il est **hors de question de réaliser la synchronisation** des processus à l'aide de la fonction **semctl**. Par exemple, un code du genre

```
...
while (semctl(idSem, 2, GETVAL) == 0)
{
}
...
```

pour attendre que le sémaphore no 2 passe à 0 est **à proscrire**. Nous sommes en effet en présence d'une **boucle infinie**, le processus ne s'endort pas, il boucle à l'infini jusqu'au moment où le sémaphore redevient positif. D'un point de vue logique pure, cela fonctionne, mais d'un point de vue programmation, c'est aberrant car le processus utilise du temps processeur pour ne rien faire réellement. La bonne manière de faire est d'utiliser la fonction **semop** décrite ci-dessous.

6.5.5 Synchroniser les processus : la fonction `semop`

Quel que soit le nombre de sémaphores d'un ensemble, un processus peut se **synchroniser sur un, plusieurs ou tous les sémaphores de l'ensemble**. Il peut donc réaliser un **nombre d'opérations** (incrémentation, décrémentation, attente de passage par 0) qui n'est **pas nécessairement égal au nombre de sémaphores** de l'ensemble. Pour cela, il doit utiliser la fonction :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

où

- **semid** est l'**identifiant** de l'ensemble de sémaphores que l'on désire manipuler,
- **sops** est le pointeur vers le premier élément d'un **vecteur** (de **structure sembuf**) contenant les **opérations** à réaliser sur l'ensemble de sémaphores.
- **nsops** est le **nombre d'opérations à réaliser atomiquement** (soit elles réussissent toutes, soit aucune ne réussit) sur l'ensemble de sémaphores. Il s'agit de la taille du vecteur pointé par **sops**.

La structure **sembuf** est définie par :

```
struct sembuf
{
    unsigned short  sem_num;      /* Numéro du sémaphore */
    short           sem_op;       /* Opération sur le sémaphore */
    short           sem_flg;      /* Options pour l'opération */
};
```

où

- **sem_num** est le numéro du sémaphore (son indice dans le vecteur pointé par **sem_base**) sur lequel on veut agir.
- **sem_op** est l'opération que l'on souhaite réaliser sur le sémaphore (ce que nous avons appelé “la demande” ci-dessus) :
 - **si $sem_op > 0$** : demande d'**incrémentation** du sémaphore de la valeur sem_op ; cette opération ne rend **jamais** l'appel de la fonction `semop` **bloquant**.
 - **si $sem_op < 0$** : demande de **décrémentation** du sémaphore de la valeur $|sem_op|$; cette opération est **bloquante** si $|sem_op|$ est plus grand que la valeur actuelle du sémaphore (`semval` de la structure `sem`).
 - **si $sem_op = 0$** : demande d'**attente** que le sémaphore passe à **0**.
- **sem_flg** est un paramètre entier permettant de modifier le comportement standard de la fonction `semop` :
 - **si $sem_flg = 0$** : il s'agit du **comportement standard**. Si l'ensemble des demandes du vecteur `sops` ne peuvent pas être acceptées, l'appel de la fonction est bloquant jusqu'au moment où celles-ci sont acceptées. Il s'agit d'une **tentative de prise du sémaphore bloquante**. De plus, le système ne mémorise pas les actions réalisées sur les sémaphores.
 - **si $sem_flg = IPC_NOWAIT$** : si l'ensemble des demandes du vecteur `sops` ne peuvent pas être acceptées, l'appel de la fonction n'est plus bloquant, elle retourne directement **-1** et `errno` est positionné à `EAGAIN`. Il s'agit d'une **tentative de prise du sémaphore non bloquante**.
 - **si $sem_flg = SEM_UNDO$** : le système mémorise les actions réalisées sur le sémaphore. En cas de mort prématurée du processus, le système peut alors “restaurer” le sémaphore, c'est-à-dire “inverser” les opérations réalisées par le processus sur le sémaphore, afin d'éviter une situation de blocage (“deadlock”).

Il est également possible de combiner par un OU logique bit à bit les valeurs `IPC_NOWAIT` et `SEM_UNDO`.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,

- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	Le processus appelant n'a pas les droits d'accès suffisant sur l'ensemble de sémaphores d'identifiant semid .
EINVAL	nsops ou semid ont une valeur illégale.
EFAULT	sops pointe en dehors de l'espace d'adressage autorisé.
EAGAIN	Une opération ne pouvait pas être effectuée immédiatement et IPC_NOWAIT a été indiqué dans sem_flg ,
ERANGE	sem_op + semval est supérieur à SEMVMX (la valeur maximale de semval autorisée par l'implémentation) pour l'une des opérations.
...	...

Exemple 6.20 (semop). Soit le programme suivant :

```
***** TestSemop.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc,char* argv[])
{
    int idSem;
    key_t cle;
    int ret;
    struct sembuf operations[2];

    cle = atoi(argv[1]);

    // Recuperation de l'ensemble de semaphores
    if ((idSem = semget(cle,0,0)) == -1)
    {
        perror("Erreur de semget");
        exit(1);
    }

    printf("idSem = %d\n",idSem);

    // Premiere synchronisation
    operations[0].sem_num = 0;
    operations[0].sem_op = -5;
    operations[0].sem_flg = 0;

    operations[1].sem_num = 2;
    operations[1].sem_op = +10;
    operations[1].sem_flg = 0;

    if (semop(idSem,operations,2) == -1)
        perror("Erreur de semop (1)");

    // Deuxieme synchronisation
```

```
operations[0].sem_num = 0;
operations[0].sem_op = -10;
operations[0].sem_flg = IPC_NOWAIT;

operations[1].sem_num = 2;
operations[1].sem_op = -15;
operations[1].sem_flg = IPC_NOWAIT;

if (semop(idSem,operations,2) == -1)
    perror("Erreur de semop (2)");

// Troisieme synchronisation
operations[0].sem_num = 0;
operations[0].sem_op = -10;
operations[0].sem_flg = 0;

operations[1].sem_num = 2;
operations[1].sem_op = -15;
operations[1].sem_flg = 0;

if (semop(idSem,operations,2) == -1)
    perror("Erreur de semop (3)");

exit(0);
}
```

dont un exemple d'exécution fournit

```
# ipcs -s -i 163840

Tableaux de sémaphores semid=163840
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Fri Aug 6 10:22:52 2021
ctime = Fri Aug 6 10:23:22 2021
semnum  valeur  ncount  zcount  PID
0        10      0        0        6041
1        20      0        0        6041
2        30      0        0        6041

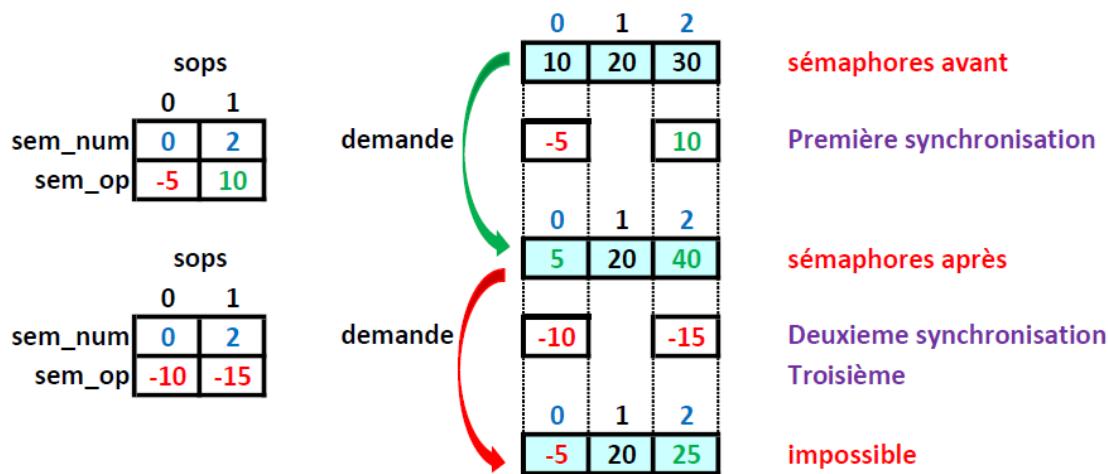
# ./TestSemop 1234 &
[1] 6080
idSem = 163840
Erreur de semop (2): Resource temporarily unavailable
# ps
PID  TTY      TIME      CMD
6080  pts/1    00:00:00 TestSemop
6087  pts/1    00:00:00 ps
10969 pts/1    00:00:01 bash
# ipcs -s -i 163840

Tableaux de sémaphores semid=163840
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600, access_perms=0600
nsems = 3
otime = Fri Aug 6 10:23:40 2021
ctime = Fri Aug 6 10:23:22 2021
semnum  valeur  ncount  zcount  PID
0        5       1        0        6080
1        20      0        0        6041
2        40      0        0        6080

#
```

On constate que :

1. Avant exécution du programme, les sémaphores de l'ensemble ont les valeurs [10][20][30].
2. Ce programme récupère tout d'abord l'identifiant de l'ensemble de sémaphores dont la **clé** est précisée en **premier argument** de la ligne de commande.
3. La **structure sembuf** n'a pas besoin d'être définie explicitement par le programmeur, elle est déjà définie dans un des fichiers headers. Un vecteur de 2 opérations (la variable **operations**) est utilisé pour agir sur 2 sémaphores de l'ensemble de 3 sémaphores.
4. Le **premier appel** à **semop** n'est pas bloquant, la demande est acceptée dans sa globalité et les sémaphores sont mis à jour. Ceci est illustré schématiquement par



5. Le **second appel à semop** retourne -1 car **la demande n'a pas pu être acceptée dans son ensemble** (bien que le sémaphore no 2 pourrait être ajusté à 25, le sémaphore no 0 ne peut pas être ajusté à -5, la demande globale est donc refusée ; il s'agit bien d'une **opération atomique**) et le flag `sem_flg` a la valeur **IPC_NOWAIT**. Ceci est indiqué par "**Resource temporarily unavailable**".
6. Le **dernier appel à semop** est **bloquant** car c'est la même demande que dans l'appel précédent mais `sem_flg` est à **0**, ce qui correspond au comportement standard de `semop`, c'est-à-dire une **tentative de prise bloquante du sémaphore**.
7. Le processus TestSemop de PID **6080** est donc bloqué sur le 3ème appel système `semop`. Ceci se voit par l'appel à la commande **ps**, mais surtout par le dernier appel à **ipcs** qui montre qu'un processus est en attente que le sémaphore no 0 ait une valeur positive suffisante (**ncount = 1**).

■

6.5.6 Fonctions simplifiées de synchronisation

De nombreuses applications ne manipulent qu'**un sémaphore à la fois**. L'accès à une seule ressource commune, comme une mémoire partagée par exemple, ne nécessite en effet qu'un seul sémaphore ne pouvant prendre que les valeurs **1** (la ressource est disponible) et **0** (la ressource n'est pas disponible). Dès lors, on définit souvent des **fonctions simplifiées de synchronisation** pour ce cas de figure. Un exemple de tels fonctions est fourni dans les fichiers **MySemaphores.h** et **MySemaphores.cpp** suivants :

```
***** MySemaphores.h *****
#ifndef MY_SEMAPHORES_H
#define MY_SEMAPHORES_H

int sem_wait(int num);
int sem_signal(int num);

#endif
```

et

```
***** MySemaphores.cpp *****
#include "MySemaphores.h"
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

extern int idSem;

int sem_wait(int num)
{
    struct sembuf action;
    action.sem_num = num;
    action.sem_op = -1;
    action.sem_flg = SEM_UNDO;      permet de ne pas faire de deadlock
    return semop(idSem,&action,1);
}

int sem_signal(int num)
{
    struct sembuf action;
    action.sem_num = num;
    action.sem_op = +1;
    action.sem_flg = SEM_UNDO;
    return semop(idSem,&action,1);
}
```

On remarque que :

1. la variable **idSem** est déclarée en **globale** et en **extern**. Cela nécessite donc que le programme qui utilisera ces fonctions va devoir définir en global cette variable qui va contenir l'**identifiant de l'ensemble de sémaphores** à manipuler.
2. ces deux fonctions peuvent agir sur un **ensemble de plusieurs sémaphores** mais sur **un seul à la fois**, celui dont le numéro est passé en paramètre.
3. la fonction **sem_wait** tire son nom du fait qu'elle est **potentiellement bloquante** vu qu'elle souhaite décrémenter le sémaphore de 1, tandis que la fonction **sem_signal** tire son nom du fait qu'elle va **"signaler"** ("notifier") aux autres processus que la ressource protégée est de nouveau disponible.
4. ces deux fonctions retournent simplement la valeur de retour de la fonction **semop**. Le test de la valeur de retour peut donc se réaliser exactement comme pour la fonction **semop**.
5. le flag **SEM_UNDO** est positionné, cela permet d'assurer qu'en cas de fin impromptue du processus, alors qu'il possède un sémaphore, le noyau en restituera l'état initial.

Ces deux fonctions sont donc fournies à titre d'exemple mais rien n'empêche d'en créer d'autres. Par exemple, si on imagine un ensemble de sémaphores ne contenant qu'un seul sémaphore binaire, on peut imaginer de programmer les mêmes fonctions mais sans la variable **extern idSem**, cet identifiant étant passé en paramètre aux fonctions tandis que le numéro de sémaphore est fixé à 0.

6.5.7 Exemple : Accès synchronisé à une mémoire partagée

Nous allons illustrer l'utilisation d'un **sémaphore binaire** (on pourrait l'appeler "**mutex**") pour protéger l'accès à **une mémoire partagée**. Dans l'exemple qui suit :

- Un processus va créer une mémoire partagée **structurée** contenant
 - une **chaîne de caractères** contenant successivement "aaaaaa", "bbbbbb", "cccccc", ...

- un **entier** qui sera incrémenté à chaque modification complète
- Ce processus va mettre **6 secondes pour modifier** complètement la chaîne de caractères et faire un **sleep** de 1 seconde entre chaque lettre.
- Ce processus va également créer et initialiser un **sémaphore** à 1 qui va donc servir à protéger l'accès à la mémoire partagée.
- Ce processus va créer un processus **fils** dont la vie sera de lire et **afficher le contenu de la mémoire partagée** à intervalles de temps aléatoires.

Afin de mettre en évidence le problème de synchronisation, deux exécutions seront réalisées, une sans l'utilisation du sémaphore et l'autre avec.

Exemple 6.21 (Accès synchronisé à une mémoire partagée). Soit le programme suivant :

```
***** TestSynchroShm.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "MySemaphores.h"

typedef struct
{
    int val;
    char message[10];
} MEM;

int idSem; // nécessaire pour utiliser MySemaphores.h

int main(int argc,char* argv[])
{
    int idShm;
    int idFils;
    MEM *p;

    // Creation de la memoire partagee
    if ((idShm = shmget(IPC_PRIVATE,sizeof(MEM),IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        perror("Erreur de shmget");
        exit(1);
    }

    printf("idShm = %d\n",idShm);

    // Attachement a et initialisation de la memoire partagee
    if ((p = (MEM*)shmat(idShm,NULL,0)) == (MEM*)-1)
    {
        perror("Erreur de shmat");
        exit(1);
    }
}
```

```
p->val = 1;
strcpy(p->message,"aaaaaaaa");

// Creation et initialisation de l'ensemble de semaphores
if ((idSem = semget(IPC_PRIVATE,1,IPC_CREAT | IPC_EXCL | 0600)) == -1)
{
    perror("Erreur de semget");
    exit(1);
}

printf("idSem = %d\n",idSem) ;

if (semctl(idSem,0,SETVAL,1) == -1)
{
    perror("Erreur de semctl");
    exit(1);
}

// Creation d'un processus fils
if ((idFils = fork()) == -1)
{
    perror("Erreur de fork");
    exit(1);
}

if (idFils == 0)
{
    // Fils
    while(1)
    {
        sleep(rand()%10);

        //sem_wait(0);
        printf("(Fils) Lecture Shm (%d) : --%s--\n",p->val,p->message);
        //sem_signal(0);
    }
}

// Pere
while(1)
{
    sleep(5);

    //sem_wait(0);
    printf("(Pere) Debut ecriture Shm (%d) : --%s--\n",p->val,p->message);
    for (int i=0 ; i<8 ; i++)
    {
        p->message[i]++;
        sleep(1);
    }
    p->val++;
    printf("(Pere) Fin ecriture Shm (%d) : --%s--\n",p->val,p->message);
    //sem_signal(0);
}
```

dont un exemple d'exécution fournit

```
# ./TestSynchroShm
idShm = 203653616
idSem = 557068
(Fils) Lecture Shm (1)      : --aaaaaaaa--
(Pere) Debut écriture Shm (1) : --aaaaaaaa--
(Fils) Lecture Shm (1)      : --bbbbbaaa--
(Pere) Fin écriture Shm (2)  : --bbbbbbbb--
(Fils) Lecture Shm (2)      : --bbbbbbbb--
(Pere) Debut écriture Shm (2) : --bbbbbbbb--
(Fils) Lecture Shm (2)      : --cccbbbbb--
(Fils) Lecture Shm (2)      : --ccccccbb--
(Pere) Fin écriture Shm (3)  : --cccccccc--
(Fils) Lecture Shm (3)      : --cccccccc--
(Pere) Debut écriture Shm (3) : --cccccccc--
(Fils) Lecture Shm (3)      : --ddddcccc--
^C
#
```

On constate que :

1. La mémoire partagée et l'ensemble de sémaphores ont été créés avec le flag **IPC_PRIVATE** à la place d'une valeur précise de clé. Cela est possible dans cet exemple car les processus qui vont utiliser ces deux IPC sont **parents**, le **fils hérite directement des identifiants d'IPC** du père. Il aurait également été possible de choisir une valeur de clé particulière sans rien changer d'autre au programme.
2. Les **fonctions de synchronisation simplifiées** de la section précédente sont utilisées, l'identifiant **idSem** de l'ensemble de sémaphores a donc été déclaré en **global** pour que ces fonctions puissent y avoir accès.
3. Il y a bel et bien un **problème de synchronisation**. Rien n'empêche le processus fils d'aller lire la mémoire partagée entre le début et la fin de sa modification par le processus père. Il en résulte des données incohérentes par moment lors de la lecture : "bbbbbaaa", "cccbbbbb", ... La **modification** de la mémoire partagée n'est donc **pas atomique**.

Si on décommente les appels à **sem_wait** et **sem_signal**, un autre exemple d'exécution fournit :

```
# ./TestSynchroShm
idShm = 205160948
idSem = 622606
(Fils) Lecture Shm (1)      : --aaaaaaaa--
(Pere) Debut écriture Shm (1) : --aaaaaaaa--
(Pere) Fin écriture Shm (2)  : --bbbbbbbb--
(Fils) Lecture Shm (2)      : --bbbbbbbb--
(Pere) Debut écriture Shm (2) : --bbbbbbbb--
(Pere) Fin écriture Shm (3)  : --cccccccc--
(Fils) Lecture Shm (3)      : --cccccccc--
(Pere) Debut écriture Shm (3) : --cccccccc--
(Pere) Fin écriture Shm (4)  : --dddddddd--
(Fils) Lecture Shm (4)      : --dddddddd--
(Fils) Lecture Shm (4)      : --dddddddd--
^C
#
```

On constate que :

1. Le **sémaphore** a été initialisé à **1**, ce qui est nécessaire, avec la fonction **semctl**. Remarquez que semctl fonctionne également sans utiliser l'union semun.
2. Les fonctions **sem_wait** et **sem_signal** ont été utilisées avec le paramètre **0** qui correspond au numéro du seul sémaphore de l'ensemble.
3. Il n'y a **plus aucun souci de synchronisation** à présent. Le sémaphore empêche le processus fils de lire la mémoire partagée tant que le processus père n'a pas entièrement modifié la chaîne de caractères. La **modification** est donc bien à présent une opération **atomique**.
4. Bien que le processus fils ne fait que lire la mémoire partagée, il est obligé de jouer le jeu en utilisant les fonctions de synchronisation, sinon le problème de synchronisation persisterait.



Chapitre 7

Les tubes de communications

7.1 Introduction

Les IPC (et plus particulièrement les files de messages) vus au chapitre précédent ne sont pas les seuls moyens de communication des processus. La technique abordée ici a été introduite dès les premières versions d'Unix, il s'agit des tubes de communication. Par comparaison :

- les **files de messages** permettent de transmettre des **données structurées** entre processus ayant des liens de parenté ou non.
- les **tubes de communication** sont utilisés pour transmettre un **flux de bytes en mode continu**.

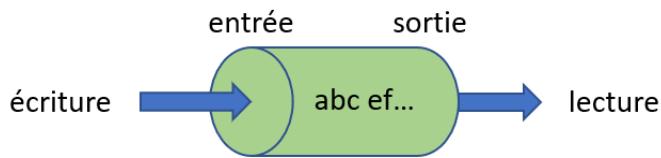
Il existe **2 types de tubes** de communication :

1. les **tubes nommés**, destinés à la communication entre processus n'ayant aucun lien de parenté. Ces tubes s'apparentent aux fichiers classiques (qui possèdent donc un nom) et ne seront pas abordés ici.
2. les **tubes anonymes**, ou **non nommés**, ou plus simplement les **tubes** (ou **pipes** en anglais), destinés à la communication entre processus faisant partie de la même famille (père, fils, petits-fils, ...). Il sont le sujet de ce chapitre.

7.2 Mécanisme et caractéristiques d'un tube

Un tube

- est un mécanisme de communication **unidirectionnel** permettant le transfert d'un flot continu de caractères (ou octets),
- est géré comme une file **FIFO**,
- possède **2 extrémités**, une pour y écrire (l'**entrée** du tube) et l'autre pour y lire (la **sortie** du tube) :



Les données écrites dans le tube sont destinées à un seul processus et ne sont donc lues qu'une seule fois. Il peut y avoir plusieurs processus qui lisent dans le même tube mais ils ne liront jamais les mêmes caractères car une fois lu, un caractère est supprimé du tube : on dit que la **lecture** dans un tube est **destructrice**.

Les tubes de communication font partie du **système de fichiers**. A chaque extrémité d'un tube correspond une **entrée dans la table des fichiers ouverts** : une pour la lecture et l'autre pour l'écriture. Un tube est donc représenté par deux entrées distinctes dans la table des fichiers ouverts.

Vu qu'il fait partie du système de fichiers, la lecture et l'écriture dans un tube se réalisent à l'aide des fonctions **read** et **write** déjà utilisées pour les fichiers non bufferisés. Par contre, l'appel à la fonction **Iseek** n'est **pas autorisée**. En effet, les caractères sont lus **une** seule fois et dans l'ordre d'entrée dans le tube. Il existe néanmoins, par rapport aux fichiers, quelques variantes dans le comportement des fonctions **read** et **write**. Celles-ci peuvent en effet devenir **bloquantes** si le tube est plein (il a donc une taille limitée mais cela reste transparent aux processus qui l'utilisent) ou s'il n'y a rien à lire dans le tube.

7.3 Crédit d'un tube : la fonction pipe

La création d'un tube (anonyme) se fait à l'aide de la fonction :

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

où **pipefd** est un vecteur qui va recevoir les descripteurs vers la sortie et l'entrée du tube. Plus précisément,

- **pipefd[0]** est le descripteur associé à la **sortie** du tube, descripteur sur lequel une **lecture** doit être réalisée,
- **pipefd[1]** est le descripteur associé à l'**entrée** du tube, descripteur sur lequel une **écriture** doit être réalisée.

La fonction **pipe** reçoit donc en paramètre un vecteur de 2 int qui n'a pas besoin d'être initialisé. En effet, la fonction **pipe**

1. crée le tube,
2. crée les deux entrées correspondantes dans la **table des fichiers ouverts** et,
3. leur attribue à chacune une valeur de descripteur, valeurs qui seront placées dans le vecteur **pipefd**.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EMFILE	Le nombre maximal de fichiers ouverts par le processus est atteint.
ENFILE	Le nombre maximal de fichiers ouverts sur le système est atteint.
EFAULT	pipefd pointe sur une mauvaise adresse.

Exemple 7.1 (pipe). Soit le programme suivant :

```
***** CreationPipe.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int fd[2];

    // Creation du pipe
    if (pipe(fd) == -1)
    {
        perror("Erreur de pipe");
        exit(1);
    }

    printf("Creation pipe OK.\n");
    printf("fd[0] = %d\n", fd[0]);
    printf("fd[1] = %d\n", fd[1]);

    // temporisation
    sleep(10);

    // Fermeture du pipe
    if (close(fd[0]) == -1)
    {
        perror("Erreur fermeture sortie du pipe");
        exit(1);
    }

    if (close(fd[1]) == -1)
    {
        perror("Erreur fermeture entree du pipe");
        exit(1);
    }

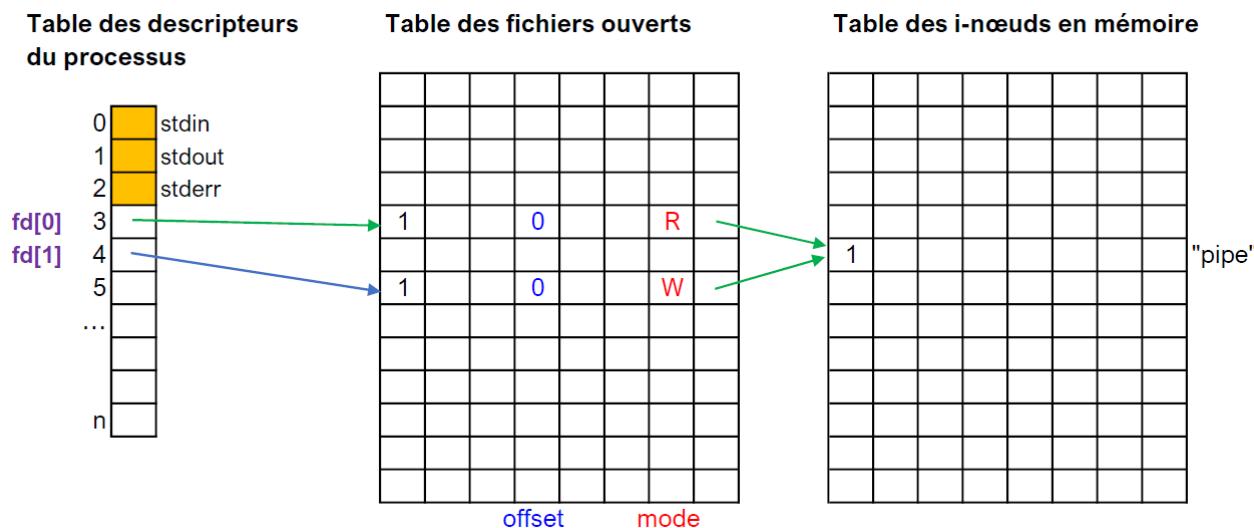
    exit(1);
}
```

dont un exemple d'exécution fournit

```
# ./CreationPipe &
[1] 18528
Creation pipe OK.
fd[0] = 3
fd[1] = 4
# lsof -p 18528 -ad "0-10"
COMMAND      PID  USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
CreationP 18528 student   0u    CHR   136,4      0t0      7 /dev/pts/4
CreationP 18528 student   1u    CHR   136,4      0t0      7 /dev/pts/4
CreationP 18528 student   2u    CHR   136,4      0t0      7 /dev/pts/4
CreationP 18528 student   3r  FIFO    0,12     0t0  144655 pipe
CreationP 18528 student   4w  FIFO    0,12     0t0  144655 pipe
# lsof -p 18528 -ad "0-10"
[1]+  Termine 1 ./CreationPipe
#
```

On constate que :

1. La création du pipe a provoqué la création de 2 entrées dans la table des fichiers ouverts. Celles-ci ont reçu comme valeurs de descripteur les valeurs **3** et **4**, **premières entrées libres** dans la table des descripteurs du processus CreationPipe :



Un **i-noeud** a également été créé, mais uniquement en mémoire, celui-ci ne sera jamais écrit sur disque.

2. Le descripteur 3 est associé à une entrée en **lecture (3r)**, il correspond à la **sortie** du tube.
3. Le descripteur 4 est associé à une entrée en **écriture (4w)**, il correspond à l'**entrée** du tube.
4. Le type des descripteurs 3 et 4 est "**FIFO**" qui correspond bien à un tube de communication.
5. Pour fermer proprement le tube, il a été nécessaire de faire appel à la fonction **close** pour les 2 descripteurs associés au tube.



7.4 Ecriture/Lecture dans un tube

Comme déjà mentionné, l'écriture et la lecture dans un tube se réalisent à l'aide des fonctions suivantes :

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

déjà abordées dans le chapitre sur les fichiers. Cependant :

- dans le cas de la fonction **write**, **fd** est le descripteur associé à l'entrée du tube. Si ce n'est pas le cas, la fonction retournera **-1** et **errno** sera positionné à **BADF**.
- dans le cas de la fonction **read**, **fd** est le descripteur associé à la sortie du tube. Si ce n'est pas le cas, la fonction retournera **-1** et **errno** sera positionné à **BADF**.

Par rapport aux fichiers, leur comportement est quelque peu différent et est conditionné par

- le **nombre d'écrivains** : nombre de descripteurs associés à l'**entrée** du tube, il s'agit du compteur situé dans l'entrée en écriture associée au tube dans la table des fichiers ouverts.
- le **nombre de lecteurs** : nombre de descripteurs associés à la **sortie** du tube, il s'agit du compteur situé dans l'entrée en lecture associée au tube dans la table des fichiers ouverts.

7.4.1 La fonction **write**

L'algorithme de la fonction **write** est alors le suivant :

Algorithme 7.1 La fonction **write** dans le contexte des tubes.

```
Si le nombre de lecteurs est égal à 0 (la sortie du tube est fermée)
    Alors
        Le signal SIGPIPE est envoyé au processus appelant.
    Sinon
        Le processus est bloqué jusqu'à ce que count octets soient libérés
        dans le tube (tube plein).
        Ecriture des count octets situés à l'adresse buf dans le tube.
        Retour du nombre d'octets écrits, c'est-à-dire count.
Fin Si
```

Exemple 7.2 (write - pipe). Soit le programme suivant :

```
***** TestWritePipe.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void handlerSIGPIPE(int sig);

int main()
{
```

```

int fd[2];

// Armement de SIGPIPE
printf("Armement de SIGPIPE\n");
struct sigaction A;
A.sa_handler = handlerSIGPIPE;
A.sa_flags = 0;
sigemptyset(&A.sa_mask);
sigaction(SIGPIPE,&A,NULL);

// Creation du pipe
if (pipe(fd) == -1)
{
    perror("Erreur de pipe");
    exit(1);
}
printf("Creation pipe OK.\n");
printf("fd[0] = %d\n",fd[0]);
printf("fd[1] = %d\n",fd[1]);

// Ecriture dans le pipe
int ret;
printf("Ecriture de 5 octets dans le pipe..."); fflush(stdout);
if ((ret = write(fd[1],"abcde",5)) != 5)
{
    perror("Erreur de write (1)");
    exit(1);
}
printf("OK\n");

// Fermeture de la sortie du pipe
if (close(fd[0]) == -1)
{
    perror("Erreur fermeture sortie du pipe");
    exit(1);
}
printf("Fermeture de la sortie du pipe\n");

// Ecriture dans le pipe
printf("Ecriture de 5 octets dans le pipe..."); fflush(stdout);
if ((ret = write(fd[1],"fghij",5)) != 5)
{
    perror("Erreur de write (2)");
    exit(1);
}
printf("OK\n");

exit(1);
}

void handlerSIGPIPE(int sig)
{
    printf("\nReception de SIGPIPE (%d)\n",sig);
}

```

dont un exemple d'exécution fournit

```
# ./TestWritePipe
Armement de SIGPIPE
Creation pipe OK.
fd[0] = 3
fd[1] = 4
Ecriture de 5 octets dans le pipe...OK
Fermeture de la sortie du pipe
Ecriture de 5 octets dans le pipe...
Reception de SIGPIPE (13)
Erreur de write (2): Broken pipe
#
```

On constate que :

1. La première écriture dans le tube a eu lieu alors que le nombre de lecteurs était égal à 1. Elle a donc réussi sans problème.
2. La seconde écriture dans le tube a eu lieu alors que le tube a été **fermé en lecture** et donc que le **nombre d'écrivains** était à égal à **0**. L'écriture a donc échoué et le signal **SIGPIPE** a été envoyé au processus. La fonction **write** a alors retourné **-1** et **errno** a été positionné à **EPIPE** (ce qui correspond au message "**Broken pipe**").



7.4.2 La fonction read

L'algorithme de la fonction **read** est alors le suivant :

Algorithme 7.2 La fonction **read** dans le contexte des tubes.

```
Si le tube est non vide et contient N octets
    Alors
        Lecture de nb_lus = minimum(N, count) octets dans le tube.
        Placement des nb_lus octets à l'adresse buf.
        Retour du nombre d'octets lus, c'est-à-dire nb_lus.
    Sinon
        Si le nombre de rédacteurs est égal à 0 (tube fermé en entrée - "fin de fichier")
            Alors
                Aucun octet n'est lu.
                Retour 0.
            Sinon
                Si la lecture est bloquante (cas par défaut)
                    Alors
                        Le processus est bloqué jusqu'à tube non vide.
                    Sinon
                        Retour -1 et errno positionné à EAGAIN.
                Fin Si
            Fin Si
        Fin Si
```

Comme on le voit, s'il y a quelque chose à lire dans le tube, la fonction **read** va lire

- **soit ce qu'elle demande (count octets)** si le tube contient un nombre N d'octets supérieur ou égal à count,

- soit la totalité des N octets présents dans le tube si celui-ci contient moins d'octets que le nombre count d'octets demandés.

On remarque également que

- s'il n'y a rien dans le tube mais qu'il reste au moins un écrivain, alors la fonction read est bloquante jusqu'au moment où un autre processus aura écrit dans le tube.
- il est possible de rendre la lecture non bloquante en cas de tube vide. Il faut pour cela réaliser l'appel de fonction suivant :

```
fcntl(fd[0], F_SETFL, O_NONBLOCK);
```

où, sans entrer dans les détails, la fonction fcntl permet d'agir sur le comportement d'un descripteur de fichier.

Exemple 7.3 (read - pipe). Soit le programme suivant :

```
***** TestReadPipe.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int fd[2];
    char buffer[80];
    int ret;

    if (pipe(fd) == -1) { perror("Erreur de pipe"); exit(1); }

    printf("fd[0] = %d fd[1] = %d\n", fd[0], fd[1]);

    // Ecriture dans le pipe
    printf("Ecriture de 6 octet dans le pipe (wagner)\n");
    if (write(fd[1], "wagner", 6) != 6) {
        perror("Erreur de write(1)"); exit(1); }

    printf("Ecriture de 8 octet dans le pipe (quettier)\n");
    if (write(fd[1], "quettier", 8) != 8) {
        perror("Erreur de write(2)"); exit(1); }

    // Lecture dans le pipe
    printf("Lecture de 10 octets dans le pipe\n");
    if ((ret = read(fd[0], buffer, 10)) < 0) {
        perror("Erreur de read(1)"); exit(1); }
    buffer[ret] = '\0';
    printf("Lu (%d) : %s\n", ret, buffer);

    printf("Lecture de 10 octets dans le pipe\n");
    if ((ret = read(fd[0], buffer, 10)) < 0) {
        perror("Erreur de read(2)"); exit(1); }
    buffer[ret] = '\0';
    printf("Lu (%d) : %s\n", ret, buffer);

    // close(fd[1]);

    printf("Lecture de 10 octets dans le pipe\n");
    if ((ret = read(fd[0], buffer, 10)) < 0) {
        perror("Erreur de read(3)"); exit(1); }
```

```

buffer[ret] = '\0';
printf("Lu (%d) : %s\n", ret, buffer);

close(fd[0]);
exit(0);
}

```

dont un exemple d'exécution fournit

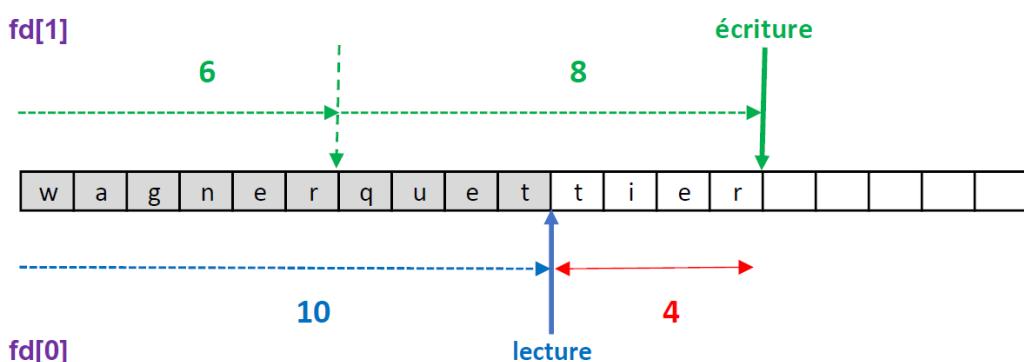
```

# ./TestReadPipe
fd[0] = 3 fd[1] = 4
Ecriture de 6 octet dans le pipe (wagner)
Ecriture de 8 octet dans le pipe (quettier)
Lecture de 10 octets dans le pipe
Lu (10) : wagnerquet
Lecture de 10 octets dans le pipe
Lu (4) : tier
Lecture de 10 octets dans le pipe
^C
#

```

On constate que :

1. Au total, $6 + 8 = 14$ octets sont écrits dans le tube avant que la première lecture n'ait lieu.
2. Lors de la **première lecture**, le tube contient $N=14$ octets et la fonction **read** demande pour lire $count=10$ octets. Le nombre d'octets lus est donc **minimum(14,10) = 10**. Après cette première lecture, le nombre d'octets restant dans le tube est égal à $14 - 10 = 4$. Ceci est représentée schématiquement par



3. Lors de la **seconde lecture**, le tube contient $N=4$ octets et la fonction **read** demande pour lire $count=10$ octets. Le nombre d'octets lus est donc **minimum(4,10) = 4**. Après cette seconde lecture, le nombre d'octets restant dans le tube est égal à $4 - 4 = 0$.
4. Lors de la **troisième lecture**, le **tube est vide** mais il **reste un écrivain** (l'entrée du tube est toujours ouverte). Le processus reste **bloqué** sur l'appel de **read** et il est nécessaire de terminer le processus par <CTRL-C>.

Si on décommente la ligne commentée dans le code ci-dessus (`close(fd[1])`), un nouvel exemple d'exécution fournit

```
# ./TestReadPipe
fd[0] = 3 fd[1] = 4
Ecriture de 6 octet dans le pipe (wagner)
Ecriture de 8 octet dans le pipe (quettier)
Lecture de 10 octets dans le pipe
Lu (10) : wagnerquet
Lecture de 10 octets dans le pipe
Lu (4) : tier
Lecture de 10 octets dans le pipe
Lu (0) :
#
```

On constate que tout le début de l'exécution est le même. Le tube ayant été fermé en écriture juste avant la dernière lecture, il n'y a **plus d'écrivain** et la fonction **read** a retourné **0**, ce qui est tout à fait équivalent à une **fin de fichier**. ■

7.5 Une communication Fils-Fils

Comme déjà mentionné, une communication par tube (anonyme) ne peut se faire qu'entre processus ayant des **liens de parenté**. En pratique, un processus crée le tube et lors de la création d'un processus fils, celui-ci va **hériter des descripteurs** de fichiers (et donc des tubes) ouverts par son père.

Dans l'exemple qui suit,

1. un processus va créer un tube,
2. il va ensuite créer 2 processus fils,
3. le premier processus fils va écrire dans le tube,
4. le second processus fils va lire dans le tube.

Exemple 7.4 (communcation fils-fils). Soit le programme suivant :

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

int Lecteur(int);
int Emetteur(int);

int main()
{
    int idFils1,idFils2;
    int fdPipe[2];

    // Creation du pipe
    printf("(pid %d) Creation du pipe\n",getpid());
    if (pipe(fdPipe)) { perror("Erreur de pipe"); exit(1); }

    printf("(pid %d) fdPipe[0] = %d fdPipe[1] : %d\n",getpid(),fdPipe[0],fdPipe[1]);
```

```
// Creation d'un 1er fils (écrivain)
if ((idFils1 = fork()) == -1) { perror("Erreur de fork (1)"); exit(1); }

if (!idFils1)
{
    // Fils 1
    close(fdPipe[0]);
    if (Emetteur(fdPipe[1]) != 0) { perror("Erreur de Emetteur"); exit(1); }
    close(fdPipe[1]);
    exit(0);
}

// Creation d'un 2ème fils (lecteur)
if ((idFils2 = fork()) == -1) { perror("Erreur de fork (2)"); exit(1); }

if (!idFils2)
{
    // Fils 2
    close(fdPipe[1]);
    if (Lecteur(fdPipe[0]) != 0) { perror("Erreur de Lecteur"); exit(1); }
    close(fdPipe[0]);
    exit(0);
}

// Fermeture du pipe par le père
close(fdPipe[0]);
close(fdPipe[1]);

// Attente de la fin des fils
wait(NULL);
wait(NULL);
exit(0);
}

int Lecteur(int fd)
{
    int ret;
    char buffer[80];

    while (1)
    {
        printf("(pid %d) Lecture de 80 octets dans le pipe\n",getpid());
        if ((ret = read(fd,buffer,80)) < 0)
            return -1;

        if (ret == 0)
        {
            printf("(pid %d) ----- Fin de lecture ---\n",getpid());
            break;
        }
        buffer[ret] = '\0';
        printf("(pid %d) Lu (%d) : %s\n",getpid(),ret,buffer);
    }
    return 0;
}

int Emetteur(int fd)
{
    printf("(pid %d) Ecriture de 14 octets dans le pipe\n",getpid());
    if (write(fd,"JeanMarcWagner",14) != 14)
```

```

    return -1;

    sleep(5);
    return 0;
}

```

dont un exemple d'exécution fournit

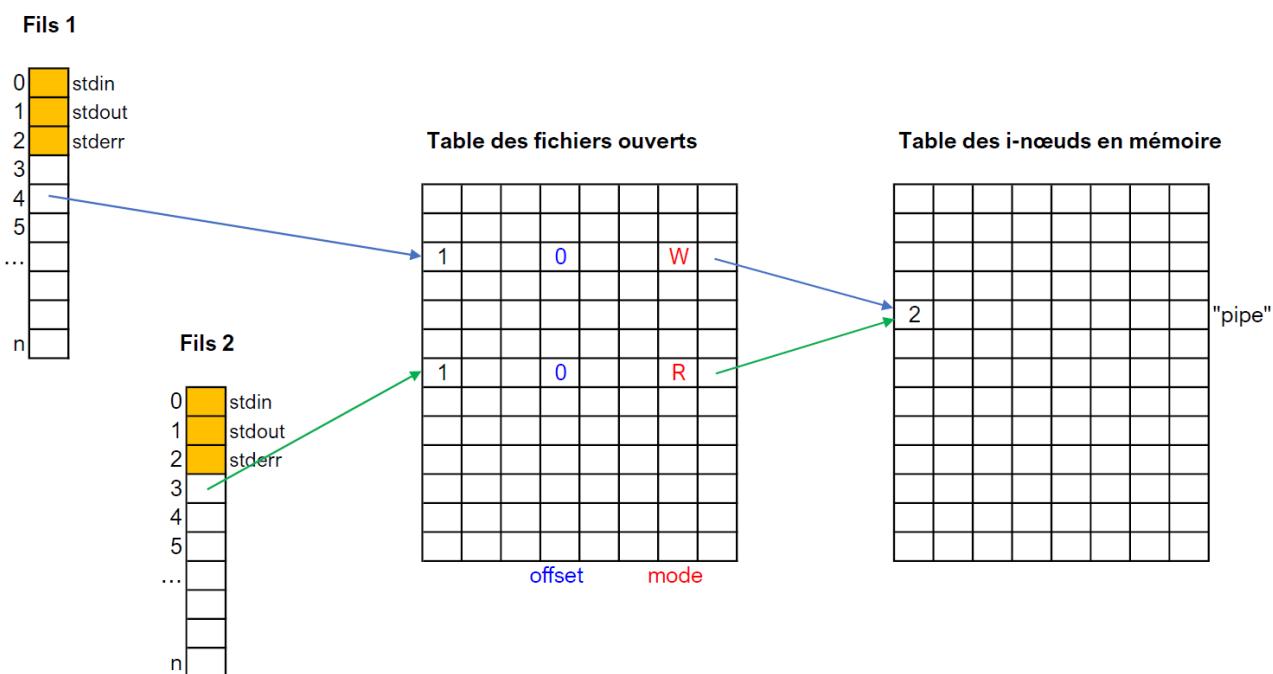
```

# ./TestPereFilsPipe
(pid 27318) Creation du pipe
(pid 27318) fdPipe[0] = 3 fPipe[1] : 4
(pid 27319) Ecriture de 14 octets dans le pipe
(pid 27320) Lecture de 80 octets dans le pipe
(pid 27320) Lu (14) : JeanMarcWagner
(pid 27320) Lecture de 80 octets dans le pipe
(pid 27320) ----- Fin de lecture ---
#

```

On constate que :

1. Après avoir créé ses deux fils, qui **héritent** des descripteurs **fdPipe** du tube, **le père ferme le tube** en entrée et en sortie. Cela n'a aucune incidence sur la communication établie entre les deux fils. Cela signifie simplement que le père ne compte pas communiquer avec ses deux fils à l'aide du tube.
2. Juste avant écriture, le **fils 1 a fermé la sortie fdPipe[0]** du tube, ce qui n'a aucune incidence vu qu'il ne fait qu'**écrire** dans le tube.
3. Juste avant lecture, le **fils 2 a fermé l'entrée fdPipe[1]** du tube, ce qui n'a aucune incidence vu qu'il ne fait que **lire** dans le tube.
4. Pendant l'exécution de la fonction **Emetteur** par le fils 1 et de la fonction **Lecteur** par le fils 2, voici la situation au niveau de la table des fichiers ouverts :



5. Le fils 1 écrit 14 octets dans le tube puis reste bloqué sur le **sleep** pendant 5 secondes. Pendant cet intervalle de temps, le **nombre d'écrivains** est égal à **1**. Dès lors, la **lecture**

est bloquante jusqu'au moment où le fils 1 sortira de sa fonction Emetteur pour ensuite fermer l'entrée du tube.

6. Le fils 2 lit les minimum(80,14) = 14 octets dans le tube lors du premier appel de read. Après être remonté dans sa boucle, le fils 2 est bloqué sur le read car le nombre d'écrivains est égal à 1. Mais cela dure 5 secondes. Dès que le fils 1 ferme l'entrée du tube, l'attente du fils 2 se termine et la fonction read retourne 0, signifiant la fin de la communication.



7.5.1 Une lecture non bloquante

Nous allons à présent refaire le même exemple que le précédent mais nous allons rendre la **lecture non bloquante** à l'aide de la fonction **fcntl**, ce qui permettrait au processus lecteur de faire autre chose tant qu'il n'y a rien à lire dans le tube. Dans l'exemple qui suit, le processus lecteur (le fils 2) se contente d'afficher "Pipe vide..." et d'attendre une seconde avant de ré-essayer de lire dans le tube.

Exemple 7.5 (communcation fils-fils - fcntl). Soit le programme suivant :

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

int Lecteur(int);
int Emetteur(int);

int main()
{
    int idFils1,idFils2;
    int fdPipe[2];

    // Creation du pipe
    printf("(pid %d) Creation du pipe\n",getpid());
    if (pipe(fdPipe)) { perror("Erreur de pipe"); exit(1); }

    printf("(pid %d) fdPipe[0] = %d fdPipe[1] : %d\n",getpid(),fdPipe[0],fdPipe[1]);

    // Creation d'un 1er fils (écrivain)
    if ((idFils1 = fork()) == -1) { perror("Erreur de fork (1)"); exit(1); }

    if (!idFils1)
    {
        // Fils 1
        close(fdPipe[0]);
        if (Emetteur(fdPipe[1]) != 0) { perror("Erreur de Emetteur"); exit(1); }
        close(fdPipe[1]);
        exit(0);
    }

    // Creation d'un 2ème fils (lecteur)
    if ((idFils2 = fork()) == -1) { perror("Erreur de fork (2)"); exit(1); }

    if (!idFils2)
    {
        // Fils 2
        if (fcntl(fdPipe[0], F_SETSIG, 0) < 0) { perror("Erreur de fcntl"); exit(1); }

        char buffer[14];
        while (read(fdPipe[0], buffer, 14) > 0)
            sleep(1);
        if (read(fdPipe[0], buffer, 14) < 0)
            perror("Erreur de read");
        else
            printf("Pipe vide...\n");
    }
}
```

```

close(fdPipe[1]);
fcntl(fdPipe[0],F_SETFL,0_NONBLOCK);
if (Lecteur(fdPipe[0]) != 0) { perror("Erreur de Lecteur"); exit(1); }
close(fdPipe[0]);
exit(0);
}

// Fermeture du pipe par le père
close(fdPipe[0]);
close(fdPipe[1]);

// Attente de la fin des fils
wait(NULL);
wait(NULL);
exit(0);
}

int Lecteur(int fd)
{
    int ret;
    char buffer[80];

    while (1)
    {
        printf("(pid %d) Lecture de 80 octets dans le pipe\n",getpid());
        if ((ret = read(fd,buffer,80)) < 0)
        {
            if (errno == EAGAIN)
            {
                printf("(pid %d) Pipe vide...\n",getpid());
                sleep(1);
                continue;
            }
            return -1;
        }

        if (ret == 0)
        {
            printf("(pid %d) ----- Fin de lecture ---\n",getpid());
            break;
        }
        buffer[ret] = '\0';
        printf("(pid %d) Lu (%d) : %s\n",getpid(),ret,buffer);
    }
    return 0;
}

int Emetteur(int fd)
{
    printf("(pid %d) Ecriture de 14 octets dans le pipe\n",getpid());
    if (write(fd,"JeanMarcWagner",14) != 14)
        return -1;

    sleep(5);
    return 0;
}

```

dont un exemple d'exécution fournit

```
# ./TestPereFilsPipeNonBloquant
(pid 28519) Creation du pipe
(pid 28519) fdPipe[0] = 3 fPipe[1] : 4
(pid 28520) Ecriture de 14 octets dans le pipe
(pid 28521) Mise en place lecture non bloquante
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Lu (14) : JeanMarcWagner
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Pipe vide...
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Pipe vide...
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Pipe vide...
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Pipe vide...
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) Pipe vide...
(pid 28521) Lecture de 80 octets dans le pipe
(pid 28521) ----- Fin de lecture ---
#
```

On constate que :

1. Le fils 2 a utilisé la fonction **fcntl** pour rendre sa **lecture non bloquante** en cas de tube vide même si le nombre d'écrivains est égal à 1.
2. La **première lecture** est similaire à l'exemple précédent. Le tube n'est pas vide et le fils 2 a **lu les 14 octets présents** dans le tube.
3. La **dernière lecture** est également similaire à l'exemple précédent. Le **tube est vide** et le **nombre d'écrivains** est égal à **0** (le fils 1 a fermé l'entrée du tube). Dès lors, la fonction **read** a retourné **0**, signifiant une fin de communication.
4. Entre les deux, le **tube est vide** et le **nombre d'écrivains** est égal à **1**. La lecture ayant été rendue non bloquante, la fonction **read** retourne **-1** et **errno** est positionné à **EAGAIN**. Dans ce cas, le fils 2 se contente d'afficher le message "Pipe vide...", d'attendre une seconde et de retenter une nouvelle lecture. Cela se fait 5 fois, tant que le fils 1 n'a pas fermé l'entrée du tube.



7.6 Redirection par tube de stdout vers stdin

Nous allons montrer ici qu'il est possible de **connecter** par tube la **sorite standard (stdout)** d'un processus à l'**entrée standard (stdin)** d'un autre processus. Cela peut se faire en programmation mais également en ligne de commande à l'aide de l' "**opérateur |**".

Comme exemple, nous allons "connecter" la commande **ls** déjà bien connue à la commande **grep** qui permet de faire une recherche de chaîne de caractères dans un fichier. Cette commande prend en paramètre la chaîne de caractères recherchée et le nom du fichier dans lequel la recherche doit se faire. Si aucun nom de fichier n'est fourni, la recherche se fait sur le **stdin** du processus.

Exemple 7.6 (grep). Voici quelques exemples d'utilisation de la commande grep :

```
# cat lignes.txt
aaaaaa
bbbbbb
ccaaac
ddddddd
eeeeef
hhhhgfg
eeeetttgg
dddggaaaaafff
# grep aaa lignes.txt
aaaaaa
ccaaac
dddggaaaaafff
# grep aaa < lignes.txt
aaaaaa
ccaaac
dddggaaaaafff
#
```

On constate que :

1. Lors du **premier appel à grep**, on a précisé la chaîne de caractères recherchée ("aaa") et le fichier ("lignes.txt") dans lequel faire la recherche. La commande grep affiche alors sur sa sortie standard les lignes du fichier contenant le mot recherché.
2. Lors du **second appel à grep**, aucun nom de fichier n'a été passé en argument. Dès lors, la recherche se fait sur l'**entrée standard** du processus grep. L'utilisation de < a connecté en ligne de commande l'entrée standard du processus au fichier "lignes.txt". Le résultat de la commande a donc été le même.



7.6.1 Connexion par programmation

Dans l'exemple qui suit,

1. un processus va créer un tube de communication, ainsi qu'un processus fils,
2. le processus **fils** va exécuter la commande **ls** en ayant **redirigé** sa **sortie standard** vers **l'entrée du tube**,
3. le processus **père** va exécuter la commande **grep** (avec en argument une chaîne de caractères à rechercher) en ayant **redirigé** son **entrée standard** vers la **sortie du tube**.

Exemple 7.7 (Redirection de stdout vers stdin par programmation). Soit le programme suivant :

```
***** RedirecStdoutStdin.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    int fdPipe[2];
```

```

int idFils;

if (argc != 2)
{
    printf("Erreur: Mauvais nombre d'arguments\n");
    printf("Usage: RedirecStdoutStdin chaine\n");
    exit(1);
}

if (pipe(fdPipe) == -1) { perror("Erreur de pipe"); exit(1); }

if ((idFils = fork()) == -1) { perror("Erreur de pipe"); exit(1); }

if (!idFils)
{
    // Fils
    if (dup2(fdPipe[1], 1) == -1) { perror("Erreur de dup2"); exit(1); }
    close(fdPipe[1]);
    close(fdPipe[0]);
    execlp("ls", "ls", NULL);
}

// Père
if (dup2(fdPipe[0], 0) == -1) { perror("Erreur de dup2"); exit(1); }
close(fdPipe[1]);
close(fdPipe[0]);
execlp("grep", "grep", argv[1], NULL);
}

```

dont un exemple d'exécution fournit

```

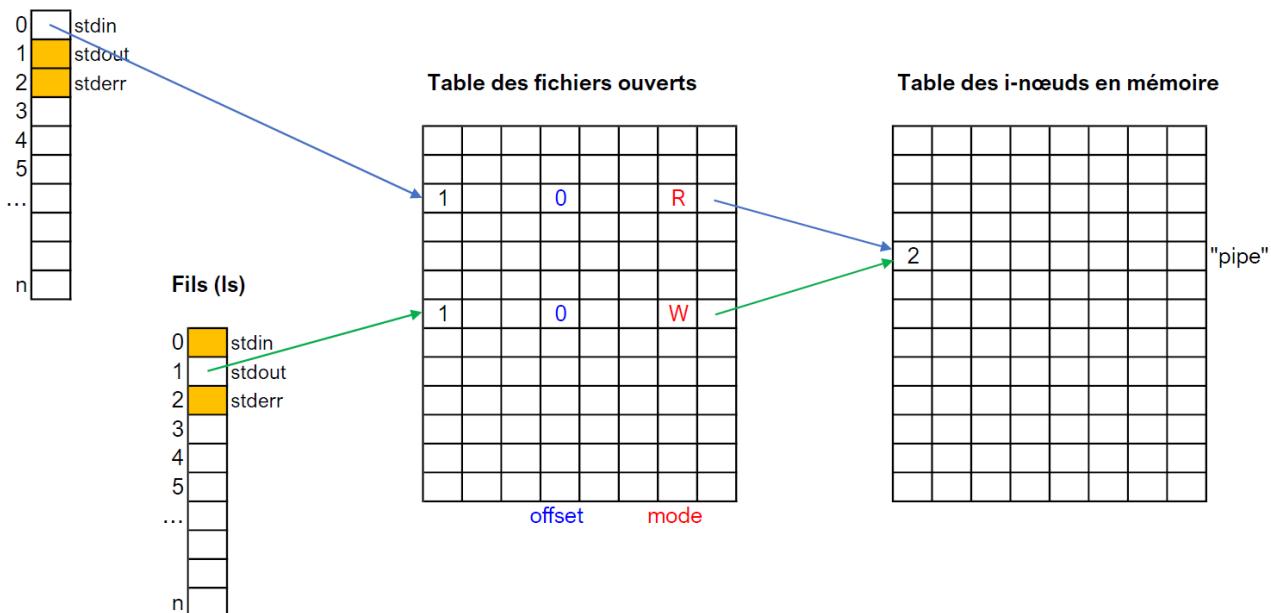
# ls
CreationPipe           TestPereFilsPipe          TestReadPipe.cpp
CreationPipe.cpp       TestPereFilsPipe.cpp      TestWritePipe
lignes.txt             TestPereFilsPipeNonBloquant   TestWritePipe.cpp
RedirecStdoutStdin    TestPereFilsPipeNonBloquant.cpp
RedirecStdoutStdin.cpp TestReadPipe
# ./RedirecStdoutStdin .cpp
CreationPipe.cpp
RedirecStdoutStdin.cpp
TestPereFilsPipe.cpp
TestPereFilsPipeNonBloquant.cpp
TestReadPipe.cpp
TestWritePipe.cpp
# ./RedirecStdoutStdin Test
TestPereFilsPipe
TestPereFilsPipe.cpp
TestPereFilsPipeNonBloquant
TestPereFilsPipeNonBloquant.cpp
TestReadPipe
TestReadPipe.cpp
TestWritePipe
TestWritePipe.cpp
#

```

On constate que :

1. La commande **ls** utilisée seule écrit sur sa **sortie standard** la liste de tous les fichiers contenus dans le répertoire courant.
2. Par l'appel à la fonction **dup2**, le processus **fils** a dupliqué le descripteur associé à **l'entrée du tube** vers le descripteur **1**, c'est-à-dire son **stdout**. Dès qu'il voudra faire une écriture sur son **stdout**, il écrira en réalité dans le tube.
3. Par l'appel à la fonction **dup2**, le processus **père** a dupliqué le descripteur associé à la **sortie du tube** vers le descripteur **0**, c'est-à-dire son **stdin**. Dès qu'il voudra faire une lecture sur son **stdin**, il lira en réalité dans le tube.
4. On a l'impression que les processus père et fils ont fermé le tube en entrée et en sortie par leurs appels à la fonction **close**. Il n'en est rien, la duplication des descripteurs assure que le tube est toujours **ouvert en lecture pour le processus père** et en **écriture pour le processus fils**. Voici la situation au niveau de la table des fichiers ouverts au moment où les processus exécutent la fonction **execvp** :

Père (grep ...)



5. C'est la fonction **execvp** qui a été utilisée ici car les commandes **ls** et **grep** doivent être recherchée dans le **PATH**.
6. Après **execvp**, **ls écrit donc à l'entrée du tube** tandis que **grep lit à la sortie du tube**. Ceci a donc pour effet combiné que grep n'affiche que les fichiers dont le nom contient la chaîne de caractères passée en paramètre, c'est-à-dire soit **.cpp** soit **Test**.



7.6.2 Connexion par ligne de commande

Il est possible de reproduire l'effet du programme de l'exemple précédent en ligne de commande. On peut connecter la sortie standard d'un processus à l'entrée standard d'un autre processus en utilisant l'"opérateur |".

Exemple 7.8 (Redirection de stdout vers stdin en ligne de commande).

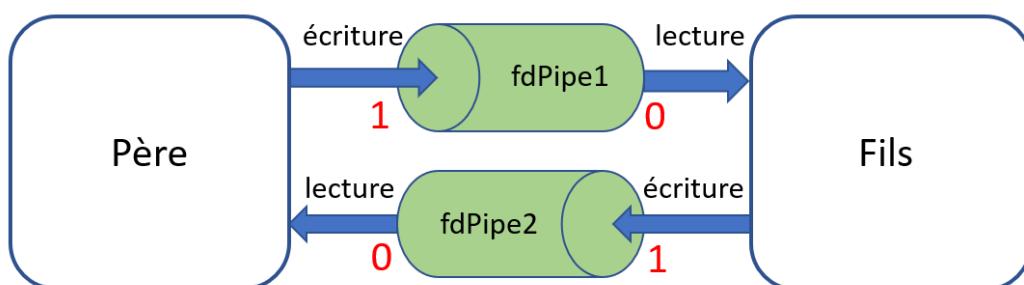
```
# ls
CreationPipe           TestPereFilsPipe          TestReadPipe.cpp
CreationPipe.cpp       TestPereFilsPipe.cpp      TestWritePipe
lignes.txt             TestPereFilsPipeNonBloquant   TestWritePipe.cpp
RedirecStdoutStdin    TestPereFilsPipeNonBloquant.cpp
RedirecStdoutStdin.cpp TestReadPipe
# ls | grep .cpp
CreationPipe.cpp
RedirecStdoutStdin.cpp
TestPereFilsPipe.cpp
TestPereFilsPipeNonBloquant.cpp
TestReadPipe.cpp
TestWritePipe.cpp
# ls | grep Test
TestPereFilsPipe
TestPereFilsPipe.cpp
TestPereFilsPipeNonBloquant
TestPereFilsPipeNonBloquant.cpp
TestReadPipe
TestReadPipe.cpp
TestWritePipe
TestWritePipe.cpp
#
#
```

On constate que le comportement est exactement le même que précédemment. La **sortie standard** de **ls** a été “connectée” à l'**entrée standard** de **grep** par tube de communication **|**.

7.7 Une communication bidirectionnelle

Nous avons vu qu'un tube de communication est un système de communication **unidirectionnelle**. Si on veut réaliser une communication bidirectionnelle entre deux processus, il est nécessaire de créer deux tubes de communication.

Dans l'exemple qui suit, un processus va créer **deux tubes** de communication **fdPipe1** et **fdPipe2**, ainsi qu'un fils avec qui il va communiquer de manière **bidirectionnelle**. Ceci est illustré par le schéma suivant :



Exemple 7.9 (Communication bidirectionnelle par tube). Soit le programme suivant :

```
***** PipesBidirectionnel.cpp *****
#include <stdio.h>
```

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

int main()
{
    int fdPipe1[2],fdPipe2[2];
    int idFils,ret;
    char buffer[80];

    if (pipe(fdPipe1) | pipe(fdPipe2))
    {
        perror("Erreur de creation d'un des pipes..."); exit(1);
    }

    if ((idFils = fork()) == -1) { perror("Erreur de fork"); exit(1); }

    if (!idFils)
    {
        // Fils
        printf("\t(FILS) Attend une demande\n");
        if ((ret=read(fdPipe1[0],buffer,80)) < 0) { perror("Erreur de read"); exit(1); }
        printf("\t(FILS) Lu (%d): %s\n",ret,buffer);
        if (write(fdPipe2[1],"Tout va bien",13) != 13) { perror("Erreur de write"); exit(1); }
        exit(0);
    }

    // Père
    printf("(PERE) J'envoie une demande \n");
    if (write(fdPipe1[1],"Wagner Jean-Marc",16) != 16) { perror("Erreur de write"); exit(1); }
    printf("(PERE) Attend la reponse...\n");
    if ((ret=read(fdPipe2[0],buffer,80)) < 0) { perror("Erreur de read"); exit(1); }
    printf("(PERE) Reponse (%d) : %s\n",ret,buffer);
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ./PipesBidirectionnel
(PERE) J'envoie une demande
(PERE) Attend la reponse...
    (FILS) Attend une demande
    (FILS) Lu (16): Wagner Jean-Marc
(PERE) Reponse (13) : Tout va bien
#
```

On constate que :

1. C'est le **fils** qui, en premier, se met en attente de lecture sur le tube **fdPipe1**.
2. Le **père** écrit à l'entrée de tube **fdPipe1** avant d'attendre une réponse à la sortie du tube **fdPipe2**.
3. Après avoir lu à l'entrée du tube **fdPipe1**, le **fils** écrit sa réponse à l'entrée du tube **fdPipe2**.



un verrou est attaché au processus, il possède une table, donc quand un processus est mort, la table des verrous part avec, contrairement au sémaphore

Chapitre 8

Compléments sur les fichiers

8.1 Les verrous

8.1.1 Principe

Nous savons à présent que plusieurs processus peuvent simultanément ouvrir et manipuler le même fichier. Il se peut donc qu'un processus écrive dans un fichier pendant qu'un autre processus lise la portion du fichier en cours de modification. Cela peut donc provoquer des incohérences dans les données lues par le second processus.

Nous savons également qu'il existe une solution radicale à ce problème : l'utilisation d'un **sémaphore**. Celui-ci pourrait donc empêcher l'accès au fichier pendant qu'un processus est en train d'écrire ou de lire dedans. Cependant, pendant que le sémaphore est acquis par un des processus :

- l'accès au fichier n'est autorisé par **aucun autre processus**, alors que plusieurs processus pourraient lire le fichier simultanément sans provoquer d'incohérence.
- **tout le fichier est "bloqué"** alors que le processus qui le manipule ne réalise peut-être qu'une modification sur une petite portion du fichier.

L'utilisation d'un sémaphore est donc un peu **trop radicale**, surtout dans le cas d'un gros fichier qui pourrait être manipulé par plusieurs processus **simultanément** mais dans des **régions différentes** du fichier. De plus, il serait intéressant que plusieurs processus puissent **lire simultanément la même portion** de fichier.

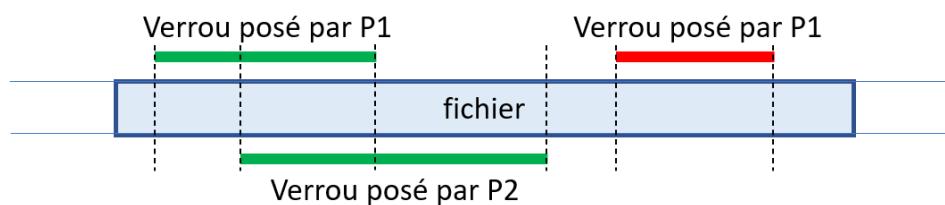
L'idée des verrous se base donc sur ces dernières constatations. Un **verrou**

- est un mécanisme qui permet de "verrouiller" une portion d'un fichier, et non obligatoirement l'entièreté du fichier,
- peut être de **2 types** :
 - **partagé** (ou "**en lecture**") : dans ce cas, **plusieurs verrous partagés** peuvent être placés **simultanément sur la même portion** de fichier. Ils permettent alors à plusieurs processus de lire en même temps la même portion de fichier.
 - **exclusif** (ou "**en écriture**") : dans ce cas, **un seul verrou exclusif peut être placé sur une**

portion de fichier. Aucun autre processus ne peut poser un verrou partagé ou un autre verrou exclusif sur cette portion du fichier. Cela permet au processus qui a posé le verrou exclusif d'écrire dans le fichier en étant certain qu'aucun autre processus n'accède à cette portion du fichier.

- est un mécanisme **attaché à l'i-noeud en mémoire du fichier ouvert** (dans la table des i-noeuds en mémoire). Les verrous ne sont donc pas écrits sur disque mais sont accessibles à tous les processus qui ont ouvert le fichier concerné.
- peut porter, comme on l'a déjà dit, sur une **partie du fichier** ou sur l'**entièreté du fichier**. Si une écriture se réalise en fin de fichier, il est possible qu'un **verrou s'allonge** en même temps que le fichier.
- est la **propriété d'un processus**. Seul le processus qui a posé un verrou peut le supprimer. Si le processus meurt avant d'avoir retiré ses verrous, tous ses verrous sont **levés automatiquement à sa mort**.

En guise d'illustration de ces concepts, voici le schéma d'un fichier sur lequel 3 verrous ont été posés :



Nous y observons

- un verrou **partagé** (en vert) posé par le processus P1,
- un verrou **exclusif** (en rouge) posé par le processus P1,
- un verrou partagé posé par le processus P2. On voit que les deux verrous partagés couvrent une portion commune du fichier, ce qui ne pose aucun problème.
- par contre, aucun autre verrou ne peut être posé sur la région déjà couverte par le verrou exclusif posé par P1.

Remarquons encore que les verrous qui vont être étudiés ici sont **consultatifs**. Cela signifie que la pose d'un verrou n'empeche pas "physiquement" un processus d'écrire ou de lire dans le fichier. Pour que le système fonctionne, les processus doivent "**jouer le jeu**" et vérifier ("consulter") qu'un verrou a été posé sur le fichier avant d'effectuer une lecture ou une écriture. Cela était déjà le cas avec les sémaphores. Par exemple, la prise d'un sémaphore par un processus n'empêche pas un autre processus de modifier le contenu de la mémoire partagée "associée".

Remarquons également que

- Pour poser un verrou **partagé** sur un fichier, un processus doit avoir ouvert le fichier en **lecture**.

- Pour poser un verrou **exclusif** sur un fichier, un processus doit avoir ouvert le fichier en **écriture**.
- Pour poser des verrous **partagés et exclusifs** sur un fichier, un processus doit avoir ouvert le fichier en **lecture/écriture**.

8.1.2 La fonction fcntl

La fonction fcntl a déjà été rencontrée lors de l'étude des tubes de communication. Dans ce contexte, elle a permis de rendre une lecture (dans un tube vide) non bloquante. En réalité, la fonction fcntl est beaucoup plus générale que cela et est une véritable boîte à outils qui permet de réaliser diverses **opérations sur les descripteurs de fichiers**. Son prototype est le suivant :

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

où

- **fd** est le descripteur du fichier ouvert sur lequel on désire agir,
- **cmd** est un entier permettant de choisir l'action à **réaliser** sur le descripteur,
- **arg** est un paramètre optionnel dépendant de l'action à réaliser.

Voici quelques commandes/actions possibles :

cmd	arg	action
F_DUPFD	une valeur de descripteur	Trouver le plus petit numéro de descripteur libre supérieur ou égal à arg et le transformer en copie de fd . Ceci est différent de dup2, qui utilise exactement le descripteur transmis. En cas de réussite, le nouveau descripteur est renvoyé.
F_GETFD	/	Récupérer les attributs du descripteur fd .
F_SETFD	une valeur entière	Positionner les attributs du descripteur de fichier avec la valeur précisée par arg . Actuellement, un seul attribut est défini : il s'agit de FD_CLOEXEC , l'attribut « close-on-exec ». Si arg = FD_CLOEXEC, le descripteur de fichier est fermé au moment d'un exec.
F_GETFL	/	Obtenir le mode d'accès et les attributs d'état du fichier.
F_SETFL	une valeur entière	Positionner les nouveaux attributs pour le descripteur de fichier à la valeur indiquée par arg . Les modes d'accès O_RDONLY, O_WRONLY, O_RDWR et les attributs de création (O_CREAT, O_EXCL, O_TRUNC de arg sont ignorés. Cette commande ne peut changer que O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME et O_NONBLOCK.
F_GETLK	adresse d'un verrou	Recuperer des informations sur d'éventuels verrous déjà posés.
F_SETLK	adresse d'un verrou	Tentative de pose non bloquante d'un verrou.
F_SETLKW	adresse d'un verrou	Tentative de pose bloquante d'un verrou.

Dans le cadre des verrous, ce sont les 3 dernières commandes qui nous intéressent. Dans ce cas, la fonction **fcntl** prend en argument l'adresse d'une variable de type verrou représenté par la structure suivante :

```
#include <fcntl.h>

struct flock
{
    ...
    short l_type;      /* Type de verrouillage : F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;   /* Interprétation de l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;    /* Décalage de début du verrouillage */
    off_t l_len;      /* Nombre d'octets du verrouillage */      0=> verrouiller verrou jusque la fin
    pid_t l_pid;      /* PID du processus bloquant notre verrou (F_GETLK) */
    ...
};
```

où

- **l_type** est le **type de verrou** que l'on veut poser :
 - **F_RDLCK** : verrou partagé (ou en lecture),
 - **F_WRLCK** : verrou exclusif (ou en écriture),
 - **F_UNLOCK** : utilisé pour retirer un verrou déjàposé.
- **l_whence** : **base** à partir de laquelle on mesure le début du verrou :
 - SEEK_SET : début du fichier,
 - SEEK_CUR : position courante de l'offset dans le fichier,
 - SEEK_END : fin du fichier.
- **l_start** : position du **début du verrou**, mesuré par rapport à **l_whence**. Cette valeur peut donc être positive, négative ou nulle selon le cas.
- **l_len** : **taille du verrou** en bytes. Cette valeur peut être positive ou nulle. Si cette valeur est **égale à 0**, le verrou est posé (ou testé selon le cas) jusqu'à la **fin du fichier** et **s'allonge avec le fichier**.
- **l_pid** : PID du processus propriétaire du verrou posé. Ce champ est automatiquement modifié par le noyau.

L'utilisation de la commande **F_SETLKW** réalise une tentative de **pose bloquante d'un verrou**. Si celui-ci ne peut être posé (car un autre verrou non compatible est déjà posé à cet endroit), l'appel de la fonction fcntl est bloquant jusqu'au moment où la pose du verrou est possible, ou si la fonction est interrompue par un signal.

L'utilisation de la commande **F_SETLK** réalise une tentative de **pose non bloquante d'un verrou**. Si celui-ci ne peut être posé (car un autre verrou non compatible est déjà posé à cet endroit), l'appel de la fonction fcntl n'est pas bloquant. La fonction retourne -1 et errno est positionné à EAGAIN.

L'utilisation de la commande **F_GETLK teste l'existence d'un verrou**, c'est-à-dire qu'elle réalise une recherche dans le fichier d'un verrou déjà posé, et cela à partir de la position précisée dans la structure flock. Dès qu'un verrou est trouvé (ou non), la structure reçue en paramètre est remplie par le noyau avec les caractéristiques du verrou trouvé.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCES	Un verrou est déjà posé par un autre processus.
EAGAIN	Un verrou est déjà posé par un autre processus.
EDEADLK	Le verrouillage avec la commande F_SETLKW conduirait à une situation d'inter-blocage.
EINT	la fonction a été interrompue par un signal.
...	

8.1.3 Pose non bloquante d'un verrou

Dans l'exemple qui suit, nous allons montrer comment réaliser la pose d'un verrou partagé/exclusif de manière **non bloquante** avec la commande **F_SETLK**.

Exemple 8.1 (Pose non bloquante d'un verrou - fcntl). Soit le programme suivant :

```
***** PoseVerrouNonBloquant.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc,char* argv[])
{
    int fd;
    struct flock verrou;
    int debut, longueur;

    if (argc != 4)
    {
        printf("Trop ou trop peu de parametre(s)\n");
        printf("Usage: PoseVerrouNonBloquant type(R/W) debut longueur\n");
        exit(1);
    }

    // Ouverture du fichier
    if ((fd = open("fichier.dat",O_RDWR)) == -1)
    {
        perror("Erreur de open");
        exit(1);
    }

    // Confection en local du verrou
    debut = atoi(argv[2]);
    longueur = atoi(argv[3]);
    if (argv[1][0] == 'W')
    {
        verrou.l_type = F_WRLCK;
        printf("(%) Tentative de pose d'un verrou exclusif de %d à %d\n",
               getpid(),debut,debut+longueur);
    }
    else
    {
        verrou.l_type = F_RDLCK;
        printf("(%) Tentative de pose d'un verrou partagé de %d à %d\n",
               getpid(),debut,debut+longueur);
    }
    verrou.l_whence = SEEK_SET;
```

```
verrou.l_start = debut;
verrou.l_len = longueur;

// Pose non bloquante du verrou
if (fcntl(fd,F_SETLK,&verrou) == -1)
{
    switch (errno)
    {
        case EACCES:
        case EAGAIN: printf("(%)d Impossible. Verrou existant...\\n",getpid());
                      break;
        default: printf("Cas non prévu.\\n");
    }
    exit(1);
}
printf("(%)d Verrou posé avec succès !\\n",getpid());

// Temporisation
sleep(20);

// Confection en local d'un verrou afin de lever le verrou posé
verrou.l_type = F_UNLCK;
verrou.l_whence = SEEK_SET;
verrou.l_start = debut;
verrou.l_len = longueur;

// Levée du verrou
if (fcntl(fd,F_SETLK,&verrou) == -1)
{
    switch (errno)
    {
        case EACCES:
        case EAGAIN: printf("Impossible. Verrou existant...\\n");
                      break;
        default: printf("Cas non prévu.\\n");
    }
    exit(1);
}

printf("(%)d Verrou levé avec succès !\\n",getpid());
close(fd);
exit(0);
}
```

en soit ne sert à rien, c'est un exemple, car exit(0) ferme d'office le verrou

dont un exemple d'exécution fournit

```
# touch fichier.dat
# ./PoseVerrouNonBloquant R 5 20 &
[1] 12423
(12423) Tentative de pose d'un verrou partagé de 5 à 25
(12423) Verrou posé avec succès !
# ./PoseVerrouNonBloquant R 10 20 &
[2] 12430
(12430) Tentative de pose d'un verrou partagé de 10 à 30
(12430) Verrou posé avec succès !
# ./PoseVerrouNonBloquant W 15 20 &
[3] 12437
(12437) Tentative de pose d'un verrou exclusif de 15 à 35
(12437) Impossible. Verrou existant...
[3]+ Termine 1 ./PoseVerrouNonBloquant W 15 20
#
(12423) Verrou levé avec succès !
(12430) Verrou levé avec succès !
[1]- Fini ./PoseVerrouNonBloquant R 5 20
[2]+ Fini ./PoseVerrouNonBloquant R 10 20
# ./PoseVerrouNonBloquant W 15 20 &
[1] 12450
(12450) Tentative de pose d'un verrou exclusif de 15 à 35
(12450) Verrou posé avec succès !
(12450) Verrou levé avec succès !
[1]+ Fini ./PoseVerrouNonBloquant W 15 20
#
```

On constate que :

1. La commande **touch** a permis de créer le fichier "fichier.dat" dont le contenu est vide.
2. Le fichier a été **ouvert en lecture/écriture**, ce qui permet de poser un verrou en lecture et/ou écriture sur le fichier.
3. Le **1er argument** de l'exécutable PoseVerrouNonBloquant permet de choisir le **type de verrou** à poser : **W** pour un verrou **exclusif**, **R** pour un verrou **partagé**. Les 2ème et 3ème arguments spécifient la position du début du verrou par rapport au début du fichier (verrou.l_whence est égal à SEEK_SET), et la longueur du verrou.
4. La **première exécution** du programme a réussi et un **verrou partagé allant de 5 à 25** a été posé sur le fichier par le processus 12423.
5. Le **seconde exécution** du programme a réussi et un **verrou partagé allant de 10 à 30** a été posé sur le fichier par le processus 12430. Bien que les deux verrous posés se superposent partiellement, cela ne pose aucun problème vu qu'il s'agit de verrous partagés.
6. La **troisième exécution** du programme a **échoué** car il s'agit d'une tentative de pose de **verrou exclusif** (de 15 à 35) sur une zone du fichier déjà couverte partiellement par un verrou partagé.
7. Lors de la troisième exécution du programme, la fonction fcntl étant utilisée avec la commande **F_SETLK** n'est **pas bloquante** et a **retourné -1**.
8. A la fin de leur attente, les deux processus 12423 et 12430 ont levé leurs verrous. Cela a permis la réussite d'une **quatrième exécution** du programme qui a, cette fois, réussi à posé un **verrou exclusif entre 15 et 35**.

8.1.4 Pose bloquante d'un verrou

Dans l'exemple qui suit (qui est tout à fait similaire au précédent mis à part pour la commande), nous montrons comment réaliser la pose d'un verrou partagé/exclusif de manière **bloquante** avec la commande **F_SETLKW**.

Exemple 8.2 (Pose bloquante d'un verrou - fcntl). Soit le programme suivant :

```
***** PoseVerrouNonBloquant.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc,char* argv[])
{
    int fd;
    struct flock verrou;
    int debut, longueur;

    if (argc != 4)
    {
        printf("Trop ou trop peu de parametre(s)\n");
        printf("Usage: PoseVerrouNonBloquant type(R/W) debut longueur\n");
        exit(1);
    }

    // Ouverture du fichier
    if ((fd = open("fichier.dat",O_RDWR)) == -1)
    {
        perror("Erreur de open");
        exit(1);
    }

    // Confection en local du verrou
    debut = atoi(argv[2]);
    longueur = atoi(argv[3]);
    if (argv[1][0] == 'W')
    {
        verrou.l_type = F_WRLCK;
        printf("(%d) Tentative de pose d'un verrou exclusif de %d à %d\n",
               getpid(),debut,debut+longueur);
    }
    else
    {
        verrou.l_type = F_RDLCK;
        printf("(%d) Tentative de pose d'un verrou partagé de %d à %d\n",
               getpid(),debut,debut+longueur);
    }
    verrou.l_whence = SEEK_SET;
    verrou.l_start = debut;
    verrou.l_len = longueur;

    // Pose bloquante du verrou
    if (fcntl(fd,F_SETLKW,&verrou) == -1)
```

```

{
    switch (errno)
    {
        case EACCES:
        case EAGAIN: printf("(%) Impossible. Verrou existant...\n",getpid());
                      break;
        default: printf("Cas non prévu.\n");
    }
    exit(1);
}
printf("(%) Verrou posé avec succès !\n",getpid());

// Temporisation
sleep(20);

// Confection en local d'un verrou afin de lever le verrou posé
verrou.l_type = F_UNLCK;
verrou.l_whence = SEEK_SET;
verrou.l_start = debut;
verrou.l_len = longueur;

// Levée du verrou
if (fcntl(fd,F_SETLK,&verrou) == -1)
{
    switch (errno)
    {
        case EACCES:
        case EAGAIN: printf("Impossible. Verrou existant...\n");
                      break;
        default: printf("Cas non prévu.\n");
    }
    exit(1);
}

printf("(%) Verrou levé avec succès !\n",getpid());
close(fd);
exit(0);
}

```

dont un exemple d'exécution fournit

```

# ./PoseVerrouBloquant W 5 20 &
[1] 13533
(13533) Tentative de pose d'un verrou exclusif de 5 à 25
(13533) Verrou posé avec succès !
# ./PoseVerrouBloquant R 10 20 &
[2] 13541
(13541) Tentative de pose d'un verrou partagé de 10 à 30
#
(13533) Verrou levé avec succès !
(13541) Verrou posé avec succès !
(13541) Verrou levé avec succès !
[1]- Fini ./PoseVerrouBloquant W 5 20
[2]+ Fini ./PoseVerrouBloquant R 10 20
#

```

On constate que :

1. La **première exécution** du programme a réussi et un **verrou exclusif allant de 5 à 25** a

été posé sur le fichier par le processus 13533.

2. Lors de la seconde **seconde exécution** du programme, le processus 13541 est resté **bloqué** sur la fonction **fcntl** car il essaie de poser un **verrou partagé** sur une zone du fichier partiellement occupée par le **verrou exclusif** précédent.
3. Après son attente, le processus 13533 a levé son verrou exclusif, ce qui a pour effet de débloquer le processus 13541 qui a enfin pu poser son **verrou partagé allant de 10 à 30**.



8.1.5 Test de l'existence de verrous

Nous allons à présent utiliser la commande **F_GETLK** pour tester l'existence de verrous posés dans un fichier. L'exemple qui suit ne tient pas compte du cas où il y aurait plusieurs verrous partagés posés sur une région commune du fichier. La recherche se réalise en effet de manière séquentielle.

Exemple 8.3 (Test de l'existence de verrous - fcntl). Soit le programme suivant :

```
***** TestVerrous.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int fd;
    struct flock verrou;

    if ((fd = open("fichier.dat", O_RDWR)) == -1)
    {
        perror("Erreur de open");
        exit(1);
    }

    printf("Lecture des verrous du fichier...\n");
    verrou.l_whence = SEEK_SET;
    verrou.l_start = 0;

    while(1)
    {
        verrou.l_type = F_WRLCK; // doit être initialisé
        verrou.l_len = 0; // Recherche jusque la fin du fichier

        if (fcntl(fd, F_GETLK, &verrou))
        {
            perror("Erreur de fcntl");
            exit(1);
        }

        switch (verrou.l_type)
        {
            case F_UNLCK: printf("\nFin de lecture.\n");
                            exit(0);

            case F_RDLCK: printf("----- Verrou partagé trouvé -----\\n");
        }
    }
}
```

```

        printf("propriétaire: %d\n",verrou.l_pid);
        printf("début: %d longueur: %d\n\n",verrou.l_start,verrou.l_len);
        break;

    case F_WRLCK: printf("----- Verrou exclusif trouvé -----");
        printf("propriétaire: %d\n",verrou.l_pid);
        printf("début: %d longueur: %d\n\n",verrou.l_start,verrou.l_len);
        break;
    }

    // Pour recommencer la recherche après le verrou trouvé.
    verrou.l_start += verrou.l_len;
}
}

```

dont un exemple d'exécution fournit

```

# ./PoseVerrouBloquant R 5 10 &
[1] 14575
(14575) Tentative de pose d'un verrou partagé de 5 à 15
(14575) Verrou posé avec succès !
# ./PoseVerrouBloquant W 15 20 &
[2] 14582
(14582) Tentative de pose d'un verrou exclusif de 15 à 35
(14582) Verrou posé avec succès !
# ./PoseVerrouBloquant R 100 20 &
[3] 14589
(14589) Tentative de pose d'un verrou partagé de 100 à 120
(14589) Verrou posé avec succès !
# ./TestVerrous
Lecture des verrous du fichier...
----- Verrou partagé trouvé -----
propriétaire: 14575
début: 5 longueur: 10

----- Verrou exclusif trouvé -----
propriétaire: 14582
début: 15 longueur: 20

----- Verrou partagé trouvé -----
propriétaire: 14589
début: 100 longueur: 20

Fin de lecture.
#
(14575) Verrou levé avec succès !
(14582) Verrou levé avec succès !
(14589) Verrou levé avec succès !
[1] Fini ./PoseVerrouBloquant R 5 10
[2]- Fini ./PoseVerrouBloquant W 15 20
[3]+ Fini ./PoseVerrouBloquant R 100 20
# ./TestVerrous
Lecture des verrous du fichier...
Fin de lecture.
#

```

On constate que :

1. Le champs **l_len** de la structure verrou est **mise à 0** avant chaque nouvel appel à la fonction fcntl afin que la recherche de verrou se réalise **jusque la fin du fichier**.
2. Lorsqu'un verrou est trouvé, ses caractéristiques sont affichées. La recherche recommence alors **juste après la verrou trouvé**. Voilà pourquoi dans cet exemple il est impossible de trouver des verrous qui seraient superposés.
3. La recherche se termine lorsque la fonction fcntl affecte la valeur **F_UNLOCK** au champ **l_type** de la variable verrou, indiquant qu'elle n'a **plus trouvé de verrou** sur le reste du fichier.



8.2 Les répertoires

Il est souvent important pour un programme d'accéder au **contenu d'un répertoire**, voir d'être capable d'en **créer** ou d'en **supprimer** un. Cela peut être utile si un programme souhaite récupérer des données dans des fichiers dont il ne connaît pas les noms à priori. La lecture dans un répertoire pourrait alors permettre de récupérer la liste des fichiers qui y sont présents.

8.2.1 Crédation d'un répertoire

La fonction à utiliser pour créer un répertoire est :

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

int mkdir(const char *pathname, mode_t mode);
```

où

- **pathname** est le **nom de répertoire** que l'on désire créer, il peut s'agir d'un chemin absolu ou relatif.
- **mode** correspond aux **droits d'accès** en lecture/écriture/exécution pour l'utilisateur, le groupe et les autres, exactement comme dans le cas d'une création avec la fonction open.

Remarquez que le mécanisme de l'**umask** est utilisé comme lors de la création des fichiers ordinaires. De plus, le **propriétaire du répertoire** correspondra au propriétaire du processus appelant la fonction, et le répertoire ne sera créé que si l'utilisateur a les droits sur le répertoire **parent** du nouveau répertoire.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,

- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCESS	Le processus n'a pas les droits sur le répertoire parent.
EEXIST	pathname existe déjà (pas nécessairement un répertoire).
EFAULT	pathname pointe en-dehors de l'espace d'adressage du processus.
ENOSPC	Pas assez d'espace disque.
...	

Exemple 8.4 (mkdir). Soit le programme suivant :

```
***** TestMkdir.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("Trop ou trop peu d'arguments !\n");
        printf("Usage: TestMkdir nomRepertoire\n");
        exit(1);
    }

    if (mkdir(argv[1], 0700) == -1)
    {
        perror("Erreur de mkdir");
        exit(1);
    }
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -l
total 16
-rwxrwxr-x. 1 student student 8648 24 nov 13:33 TestMkdir
-rw-rw-r--. 1 student student   375 24 nov 13:33 TestMkdir.cpp
# ./TestMkdir AAA
# ls -l
total 16
drwx----- 2 student student      6 24 nov 13:33 AAA
-rwxrwxr-x. 1 student student 8648 24 nov 13:33 TestMkdir
-rw-rw-r--. 1 student student   375 24 nov 13:33 TestMkdir.cpp
#
```

On constate que

1. Le nom du répertoire a été spécifié par **chemin relatif**. Dans ce cas-ci, le répertoire AAA a été créé dans le **répertoire courant**.
2. Le répertoire a été créé avec les **droits 700** ce qui correspond à **rwx-----**.



8.2.2 Suppression d'un répertoire

La fonction à utiliser pour supprimer un répertoire est :

```
#include <unistd.h>

int rmdir(const char *pathname);
```

où

- **pathname** est le **nom du répertoire** que l'on désire supprimer.

Remarquez que le répertoire doit être **vide** pour pouvoir être supprimé.

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et errno est positionné.

En cas d'erreur, les valeurs de errno les plus fréquentes sont

errno	signification
EACCESS	Le processus n'a pas les droits d'écriture dans le répertoire parent.
ENOTEMPTY	pathname contient d'autres éléments que . et ..
EFAULT	pathname pointe en-dehors de l'espace d'adressage du processus.
ENOTDIR	pathname ne représente pas un répertoire.
...	

Exemple 8.5 (rmdir). Soit le programme suivant :

```
***** TestRmdir.cpp *****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc,char* argv[])
{
    if (argc != 2)
    {
        printf("Trop ou trop peu d'arguments !\n");
        printf("Usage: TestRmdir nomReperoire\n");
        exit(1);
    }

    if (rmdir(argv[1]) == -1)
```

```
{
    perror("Erreur de rmdir");
    exit(1);
}

exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -l
total 32
drwx----- 2 student student 22 24 nov 14:10 AAA
-rwxrwxr-x 1 student student 8648 24 nov 13:33 TestMkdir
-rw-rw-r-- 1 student student 379 24 nov 13:39 TestMkdir.cpp
-rwxrwxr-x 1 student student 8648 24 nov 14:10 TestRmdir
-rw-rw-r-- 1 student student 348 24 nov 14:10 TestRmdir.cpp
# ls -l AAA
total 0
-rw-rw-r-- 1 student student 0 24 nov 14:10 fich.dat
# ./TestRmdir XXX
Erreur de rmdir: No such file or directory
# ./TestRmdir AAA
Erreur de rmdir: Directory not empty
# rm AAA/fich.dat
# ./TestRmdir AAA
# ls -l
total 32
-rwxrwxr-x 1 student student 8648 24 nov 13:33 TestMkdir
-rw-rw-r-- 1 student student 379 24 nov 13:39 TestMkdir.cpp
-rwxrwxr-x 1 student student 8648 24 nov 14:10 TestRmdir
-rw-rw-r-- 1 student student 348 24 nov 14:10 TestRmdir.cpp
#
```

On constate que

1. La **première exécution** du programme a échoué car la fonction rmdir a tenté de supprimer le répertoire **XXX** qui **n'existe pas**.
2. La **seconde exécution** du programme a échoué car le fonction rmdir a tenté de supprimer le répertoire **AAA** existant mais celui-ci était **non vide**.
3. La **dernière exécution** du programme a **réussi** étant donné que le répertoire AAA a été vidé par l'appel de la commande rm.



8.2.3 Changement de répertoire courant

Au moment où il est lancé, un processus acquiert un répertoire courant qui correspond au répertoire courant où il a été lancé. Il peut changer de répertoire courant en cours d'exécution en utilisant la fonction :

```
#include <unistd.h>
int chdir(const char *path);
```

où

- **path** est le **nom du répertoire** qui doit devenir le répertoire courant du processus appelant.

Cette fonction est l'équivalent en programmation de la commande **cd** qui permet de **changer le répertoire courant** de l'**interpréteur de commande** (qui peut être affiché par la commande `echo $PWD`).

Valeurs de retour

Cette fonction retourne

- la valeur **0** en cas de succès,
- la valeur **-1** en cas d'erreur, et `errno` est positionné.

En cas d'erreur, les valeurs de `errno` les plus fréquentes sont

errno	signification
EACCESS	Le processus n'a pas les droits sur le répertoire path .
EFAULT	path pointe en-dehors de l'espace d'adressage du processus.
ENOTDIR	path ne représente pas un répertoire.
...	

Exemple 8.6 (chdir). Soit le programme suivant :

```
***** TestChdir.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc,char* argv[])
{
    if (argc != 2)
    {
        printf("Trop ou trop peu d'arguments !\n");
        printf("Usage: TestChdir nomReperatoire\n");
        exit(1);
    }

    if (chdir(argv[1]) == -1)
    {
        perror("Erreur de chdir");
        exit(1);
    }

    if (execlp("ls","ls",-1,NULL) == -1)
    {
        perror("Erreur de execlp");
        exit(1);
    }
}
```

```
    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -l
total 48
drwxrwxr-x. 2 student student 40 24 nov 14:56 AAA
-rwxrwxr-x. 1 student student 8704 24 nov 14:55 TestChdir
-rw-rw-r--. 1 student student 441 24 nov 14:55 TestChdir.cpp
-rwxrwxr-x. 1 student student 8648 24 nov 13:33 TestMkdir
-rw-rw-r--. 1 student student 379 24 nov 13:39 TestMkdir.cpp
-rwxrwxr-x. 1 student student 8648 24 nov 14:10 TestRmdir
-rw-rw-r--. 1 student student 348 24 nov 14:10 TestRmdir.cpp
# ls -l AAA
total 0
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich1.dat
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich2.dat
# ./TestChdir AAA
total 0
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich1.dat
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich2.dat
#
```

On constate que

1. L'exécution de la fonction **chdir** a affecté au processus appelant un **répertoire courant** correspondant à l'argument argv[1] reçu en ligne de commande, c'est-à-dire **AAA**.
2. L'exécution de la commande **ls** (via **execlp**) par le processus s'est réalisée dans son **répertoire courant**, c'est-à-dire **AAA**.



8.2.4 Lecture du contenu d'un répertoire

Un répertoire est en réalité un **fichier spécial** contenant pour chaque fichier ou sous-répertoire une structure (dite opaque) variant selon le type de système de fichier. Chacune de ces structures contient

- le **nom du fichier**,
- le **numéro d'i-noeud** du fichier,
- des champs servant à la gestion interne des structures (champs non intéressants pour le programmeur applicatif).

Les fonctions qui permettent d'accéder au contenu d'un répertoire sont :

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

DIR *opendir(const char *nom);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

où

- **opendir** est la fonction qui permet d'**ouvrir un fichier répertoire** dont le nom est passé en paramètre. Le retour de cette fonction est du type **DIR** qui est une structure opaque, comparable au flux **FILE**, que l'on emploie sur des répertoires au lieu des fichiers. A la manière de fopen, la fonction opendir renvoie un pointeur NULL en cas d'échec.
- **closedir** est la fonction qui permet de **fermer le fichier répertoire**.
- **readdir** est la fonction qui permet de **lire une entrée dans le fichier répertoire** et de positionner la tête de lecture au début de l'enregistrement suivant. Si le retour de cette fonction est NULL, c'est que la fin du répertoire a été atteinte.

La fonction **readdir** retourne un pointeur vers une **structure dirent** (pour “**Directory Entry**”) représentant l'entrée qui vient d'être lue dans le répertoire. Cette structure est donnée par :

```
#include <dirent.h>

struct dirent
{
    ino_t d_ino;          /* numéro d'i-noeud */
    ...
    char d_name[256];    /* nom du fichier */
};
```

où

- **d_ino** le numéro d'i-noeud du fichier,

- **d_name** est le nom du fichier,
- ... sont des champs servant à la gestion interne, qui dépendent du système du fichier et qui ne sont d'aucune utilité pour le programmeur applicatif.

Exemple 8.7 (opendir, readdir, closedir). Soit le programme suivant :

```
***** LectureRepertoire.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc,char* argv[])
{
    DIR* dir;
    struct dirent *entree;

    if (argc != 2)
    {
        printf("Trop ou trop peu d'arguments !\n");
        printf("Usage: LectureRepertoire nomRepertoire\n");
        exit(1);
    }

    // Ouverture du fichier répertoire
    if ((dir = opendir(argv[1])) == NULL)
    {
        perror("Erreur de opendir");
        exit(1);
    }

    // Lecture séquentielle dans le répertoire
    while((entree = readdir(dir)) != NULL)
        printf("inoeud = %10d --> Nom du fichier : %s\n",entree->d_ino,entree->d_name);

    // temporisation
    sleep(20);

    // Fermeture du fichier
    if (closedir(dir) == -1)
    {
        perror("Erreur de closedir");
        exit(1);
    }

    exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -il
total 64
204976460 drwxrwxr-x. 2 student student 59 25 nov 09:47 AAA
70773471 -rwxrwxr-x. 1 student student 8872 25 nov 09:58 LectureRepertoire
70773461 -rw-rw-r--. 1 student student 810 25 nov 09:58 LectureRepertoire.cpp
70773468 -rwxrwxr-x. 1 student student 8704 24 nov 14:55 TestChdir
70773459 -rw-rw-r--. 1 student student 441 24 nov 14:55 TestChdir.cpp
70773464 -rwxrwxr-x. 1 student student 8648 24 nov 13:33 TestMkdir
70773457 -rw-rw-r--. 1 student student 379 24 nov 13:39 TestMkdir.cpp
70773466 -rwxrwxr-x. 1 student student 8648 24 nov 14:10 TestRmdir
70773458 -rw-rw-r--. 1 student student 348 24 nov 14:10 TestRmdir.cpp
# ./LectureRepertoire . &
[1] 20999
inoeud = 70773454 --> Nom du fichier : .
inoeud = 142499525 --> Nom du fichier : ...
inoeud = 70773457 --> Nom du fichier : TestMkdir.cpp
inoeud = 70773464 --> Nom du fichier : TestMkdir
inoeud = 70773458 --> Nom du fichier : TestRmdir.cpp
inoeud = 70773466 --> Nom du fichier : TestRmdir
inoeud = 70773459 --> Nom du fichier : TestChdir.cpp
inoeud = 70773468 --> Nom du fichier : TestChdir
inoeud = 204976460 --> Nom du fichier : AAA
inoeud = 70773461 --> Nom du fichier : LectureRepertoire.cpp
inoeud = 70773471 --> Nom du fichier : LectureRepertoire
# lsof -p 20999 -ad "0-10"
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
LectureRe 20999 student 0u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 20999 student 1u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 20999 student 2u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 20999 student 3r DIR 252,0 4096 70773454 /home/student/
# ls -il AAA
total 0
205023690 -rw-rw-r--. 1 student student 0 24 nov 14:56 fich1.dat
205023695 -rw-rw-r--. 1 student student 0 24 nov 14:56 fich2.dat
70773442 -rw-rw-r--. 1 student student 0 25 nov 09:47 fichier.txt
[1]+ Fini ./LectureRepertoire .
# ./LectureRepertoire AAA &
[1] 21039
inoeud = 204976460 --> Nom du fichier : .
inoeud = 70773454 --> Nom du fichier : ...
inoeud = 205023690 --> Nom du fichier : fich1.dat
inoeud = 205023695 --> Nom du fichier : fich2.dat
inoeud = 70773442 --> Nom du fichier : fichier.txt
# lsof -p 21039 -ad "0-10"
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
LectureRe 21039 student 0u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 21039 student 1u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 21039 student 2u CHR 136,5 0t0 8 /dev/pts/5
LectureRe 21039 student 3r DIR 252,0 59 204976460 /home/student/AAA
#
```

On constate que

1. L'option **-i** de la commande **ls** a permis d'afficher les **numéros d'i-noeud** des fichiers

listés.

2. La **première exécution** du programme a ouvert et lu le contenu du **répertoire courant**.
3. L'exécution de la commande **lsof** a permis de visualiser les fichiers ouverts par le processus LectureRepertoire. On voit bien qu'un **descripteur (3)** a été associé en **lecture (r)** au fichier répertoire qui a été ouvert. Ce fichier est bien un **fichier spécial de type DIR**.
4. La **seconde exécution** du programme a ouvert le **répertoire AAA** qui se trouvait dans le **répertoire courant** du processus.
5. On voit apparaître dans le listing des fichiers le répertoire courant **.** et le répertoire parent **..** qui sont des fichiers également.



8.2.5 Positionnement dans un fichier répertoire

Il est possible de se déplacer dans un fichier répertoire ouvert par la fonction **opendir**. Cependant, la fonction classique **lseek** n'est pas utilisable ici. En effet, un fichier répertoire est un **fichier d'enregistrements de taille variable** et il est uniquement possible de se positionner **entre deux enregistrements**, et cela en utilisant les fonctions suivantes :

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

void rewinddir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
long telldir(DIR *dirp);
```

où

- **rewinddir** est la fonction qui permet de se repositionner au **début** du fichier répertoire.
- **seekdir** est la fonction qui permet de se positionner à la position **loc** dans le fichier répertoire, loc doit correspondre au début d'un enregistrement dans le fichier répertoire, idéalement une valeur retournée par la fonction **telldir**.
- **telldir** retourne la **position actuelle** dans le fichier répertoire.

Exemple 8.8 (rewinddir, seekdir, telldir). Soit le programme suivant :

```
***** PositionReperatoire.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char* argv[])
{
    DIR* dir;
    struct dirent *entree;
    long pos;
```

```
if (argc != 2)
{
    printf("Trop ou trop peu d'arguments !\n");
    printf("Usage: PositionRépertoire nomRépertoire\n");
    exit(1);
}

// Ouverture du fichier répertoire
if ((dir = opendir(argv[1])) == NULL)
{
    perror("Erreur de opendir");
    exit(1);
}

// Lecture de la 1ère entrée
entree = readdir(dir);
if (entree != NULL)
    printf("inoeud = %10d --> Nom du fichier : %s\n",entree->d_ino,entree->d_name);

// Mémorisation de la position de la tête de lecture
pos = telldir(dir);

// Lecture de la 2ème entrée
entree = readdir(dir);
if (entree != NULL)
    printf("inoeud = %10d --> Nom du fichier : %s\n",entree->d_ino,entree->d_name);

// Déplacement juste avant la 2ème entrée
seekdir(dir, pos);

// Re-lecture de la 2ème entrée
entree = readdir(dir);
if (entree != NULL)
    printf("inoeud = %10d --> Nom du fichier : %s\n",entree->d_ino,entree->d_name);

// Déplacement en début du fichier
rewinddir(dir);

// Re-lecture de la 1ère entrée
entree = readdir(dir);
if (entree != NULL)
    printf("inoeud = %10d --> Nom du fichier : %s\n",entree->d_ino,entree->d_name);

// Fermeture du fichier
if (closedir(dir) == -1)
{
    perror("Erreur de closedir");
    exit(1);
}

exit(0);
}
```

dont un exemple d'exécution fournit

```
# ls -al AAA
total 4
drwxrwxr-x. 2 student student 59 25 nov 09:47 .
drwxrwxr-x. 3 student student 4096 25 nov 10:50 ..
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich1.dat
-rw-rw-r--. 1 student student 0 24 nov 14:56 fich2.dat
-rw-rw-r--. 1 student student 0 25 nov 09:47 fichier.txt
# ./PositionRepertoire AAA
inoeud = 204976460 --> Nom du fichier : .
inoeud = 70773454 --> Nom du fichier : ..
inoeud = 70773454 --> Nom du fichier : ..
inoeud = 204976460 --> Nom du fichier : .
#
#
```

On constate que

1. Après lecture du premier enregistrement, la position de la tête de lecture a été **mémorisée** dans la variable **pos** à l'aide de la fonction **telldir**.
2. Après lecture du second enregistrement, la tête de lecture a été **déplacée** à la position **pos** à l'aide de la fonction **seekdir**, ce qui correspond au début du second enregistrement qui a donc été lu une seconde fois.
3. Après cette lecture, la tête de lecture a été **repositionnée au début** du fichier répertoire à l'aide de la fonction **rewinddir**. La dernière lecture a donc fourni le premier enregistrement.



Chapitre 9

Annexes

9.1 Quelques commandes utiles

9.1.1 La commande “lsof”

lsof est une commande permettant, notamment, de lister les informations sur les fichiers ouverts par les différents processus du système.

Utilisation de base

Par défaut, **lsof** fournit la liste de tous les fichiers ouverts par tous les processus :

```
# lsof
COMMAND      PID    USER     FD   TYPE   DEVICE SIZE/OFF NODE NAME
(unknown)      0  root      cwd   VDIR    10,4      4096    2 /dev/hd4
init          1  root      cwd   VDIR    10,4      4096    2 /dev/hd4
init          1  root      0u    VREG    10,4    30456  599 /dev/hd4
syncd        1900544 root      cwd   VDIR    10,4      4096    2 /dev/hd4
syncd        1900544 root      0r    VCHR     2,2      0t0    442 /dev/null
syncd        1900544 root      1w    VCHR     2,2      0t0    442 /dev/null
syncd        1900544 root      2w    VCHR     2,2      0t0    442 /dev/null
shlap64     1966182 root      cwd   VDIR    10,4      4096    2 /dev/hd4
...
#
```

Signification des champs :

Champ	Signification
COMMAND	Nom du processus
PID	Pid du processus
USER	Propriétaire du processus
FD	Descripteur de fichier (File Descriptor)
TYPE	Type de fichier
DEVICE	
SIZE/OFF	Taille du fichier/Position actuelle (offset)
NODE	Numéro d'inode du fichier
NAME	Nom du fichier ouvert

Les différentes valeurs que peut prendre le champ FD (File Descriptor) sont les suivantes :

Valeur de FD	Signification
cwd	Répertoire courant (Current Working Directory)
txt	Fichier de texte
mem	Memory mapped file
mmap	Memory mapped device
un nombre	L'actuel descripteur de fichier. Le caractère qui suit représente le mode d'ouverture.

Le caractère représentant le mode d'ouverture peut prendre une des valeurs suivantes :

Caractère	Mode d'ouverture
r	lecture seule (read)
w	écriture seule (write)
u	lecture/écriture (read and write)

TYPE indique le type de fichier et peut prendre entre autre comme valeurs :

TYPE	Signification
REG	Fichier régulier
DIR	Fichier répertoire
FIFO	First In First Out
CHR	Fichier spécial de caractères

Processus utilisant un fichier donné

On peut spécifier le nom d'un fichier, **lsof** donnera alors tous les processus qui accède à ce fichier :

```
# lsof /usr/lib64/libc-2.17.so
COMMAND   PID  USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
pulseaudi 2548 student   mem   REG  252,0  2156264 218995 /usr/lib64/libc-2.17.so
bash      9333 student   mem   REG  252,0  2156264 218995 /usr/lib64/libc-2.17.so
Thunar    10213 student   mem   REG  252,0  2156264 218995 /usr/lib64/libc-2.17.so
evince   10231 student   mem   REG  252,0  2156264 218995 /usr/lib64/libc-2.17.so
evinced  10237 student   mem   REG  252,0  2156264 218995 /usr/lib64/libc-2.17.so
...
#
```

Lister les processus d'un utilisateur

Pour cela, on utilise l'option **-u** :

```
# lssof -u student
COMMAND  PID   USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
...
bash    28010 student  txt    REG  252,0    964592 201336048 /usr/bin/bash
bash    28010 student  mem    REG  252,0   2156264 218995 /usr/lib64/libc-2.17.so
bash    28010 student  mem    REG  252,0    19256 219001 /usr/lib64/libdl-2.17.so
bash    28010 student  mem    REG  252,0   163320 2999067 /usr/lib64/ld-2.17.so
bash    28010 student  0u    CHR   136,2      0t0      5 /dev/pts/2
bash    28010 student  1u    CHR   136,2      0t0      5 /dev/pts/2
bash    28010 student  2u    CHR   136,2      0t0      5 /dev/pts/2
bash    28010 student 255u    CHR   136,2      0t0      5 /dev/pts/2
...
#
```

Lister les fichiers ouverts par un processus

Pour cela, on utilise l'option **-p** suivie du pid du processus. Soit le programme suivant :

```
/** TestLSOF.cpp ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fd;
    printf("pid = %d\n",getpid());

    fd = open("/home/student/Essai.dat",O_RDWR|O_CREAT,0644);
    lseek(fd,10,SEEK_SET);
    write(fd,"abcde",5);
    lseek(fd,-8,SEEK_CUR);

    sleep(10); // pour permettre de faire un lssof

    close(fd);
    exit(0);
}
```

Un exemple d'exécution fournit

```
# TestLSOF &
[1] 5564
pid = 5564
# lsof -p 5564
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
TestLSOF 5564 student cwd DIR 252,0 4096 205197316 /home/student/
TestLSOF 5564 student rtd DIR 252,0 4096 128 /
TestLSOF 5564 student txt REG 252,0 8856 205197334 /home/student/TestLSOF
TestLSOF 5564 student mem REG 252,0 2156264 218995 /usr/lib64/libc-2.17.so
TestLSOF 5564 student mem REG 252,0 88720 12975393 /usr/lib64/libgcc_s-4.8.5-201...
TestLSOF 5564 student mem REG 252,0 1136952 219003 /usr/lib64/libm-2.17.so
TestLSOF 5564 student mem REG 252,0 995840 12975395 /usr/lib64/libstdc++.so.6.0.19
TestLSOF 5564 student mem REG 252,0 163320 2999067 /usr/lib64/ld-2.17.so
TestLSOF 5564 student 0u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 1u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 2u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 3u REG 252,0 15 487807 /home/student/Essai.dat
#
```

Le champ SIZE/OFF nous indique actuellement la taille du fichier, c'est-à-dire 15 bytes. Pour obtenir la position actuelle dans le fichier (l'offset), c'est-à-dire 7 bytes, on peut utiliser l'option **-o** :

```
# TestLSOF &
[1] 5564
pid = 5564
# lsof -p 5564 -o
COMMAND PID USER FD TYPE DEVICE OFFSET NODE NAME
TestLSOF 5564 student cwd DIR 252,0 4096 205197316 /home/student/
TestLSOF 5564 student rtd DIR 252,0 4096 128 /
TestLSOF 5564 student txt REG 252,0 8856 205197334 /home/student/TestLSOF
TestLSOF 5564 student mem REG 252,0 2156264 218995 /usr/lib64/libc-2.17.so
TestLSOF 5564 student mem REG 252,0 88720 12975393 /usr/lib64/libgcc_s-4.8.5-201...
TestLSOF 5564 student mem REG 252,0 1136952 219003 /usr/lib64/libm-2.17.so
TestLSOF 5564 student mem REG 252,0 995840 12975395 /usr/lib64/libstdc++.so.6.0.19
TestLSOF 5564 student mem REG 252,0 163320 2999067 /usr/lib64/ld-2.17.so
TestLSOF 5564 student 0u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 1u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 2u CHR 136,1 0t0 4 /dev/pts/1
TestLSOF 5564 student 3u REG 252,0 0t7 487807 /home/student/Essai.dat
#
```

Sélectionner les descripteurs à afficher

L'option **-d** permet de sélectionner les fichiers à afficher et dont les descripteurs sont présents dans un ensemble découpé par des virgules. Par exemple, «1,3» ou «6,cwd,2». Un intervalle de numéros de descripteurs de fichiers peut également être inclus dans l'ensemble. Par exemple, «0-7» ou «3-10».

En reprenant le même programme que ci-dessus, un exemple d'utilisation est le suivant :

```
# ./TestLSOF &
[1] 29009 pid = 29009
# lsof -p 29009 -ad "1,3"
COMMAND PID USER FD   TYPE DEVICE SIZE/OFF NODE NAME
TestLSOF 29009 student  1u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  3u  REG  252,0       15 487807 /home/student/Essai.dat
# lsof -p 29009 -ad "0,cwd"
COMMAND PID USER FD   TYPE DEVICE SIZE/OFF NODE NAME
TestLSOF 29009 student cwd   DIR  252,0       42 203289826 /home/student
TestLSOF 29009 student  0u  CHR  136,2      0t0      5 /dev/pts/2
# lsof -p 29009 -ad "0-3"
COMMAND PID USER FD   TYPE DEVICE SIZE/OFF NODE NAME
TestLSOF 29009 student  0u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  1u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  2u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  3u  REG  252,0       15 487807 /home/student/Essai.dat
# lsof -p 29009 -aod "0-3"
COMMAND PID USER FD   TYPE DEVICE  OFFSET NODE NAME
TestLSOF 29009 student  0u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  1u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  2u  CHR  136,2      0t0      5 /dev/pts/2
TestLSOF 29009 student  3u  REG  252,0      0t7 487807 /home/student/Essai.dat
#
```

Dans cet exemple, l'option **-a** ("and") permet de combiner simultanément plusieurs options.

9.1.2 La commande “tar”

La commande **tar** permet de compresser dans un seul fichier appelé “**archive**” (d’extension **.tar** la plupart du temps) une collection de fichiers et de répertoires dont elle n’altère pas les caractéristiques (leur permission et autres caractéristiques restent intactes pendant la compression). Elle est particulièrement utile pour transférer une énorme quantité de fichiers (et de répertoires) d’un serveur à l’autre.

Les options principales de cette commande sont

Option	Signification
c	crée une nouvelle archive
x	extraire d’une archive
v	affiche une description verbeuse de la progression de la compression/décompression
f	précise le nom de l’archive
t	liste le contenu d’une archive

Création d’une archive contenant un répertoire

Voici un exemple de création d’une archive AAA.tar contenant le répertoire AAA et contenant lui-même deux fichiers et un sous-répertoire BBB :

```
# ls -l
total 100
drwxrwxr-x. 3 student student 55 24 fév 11:34 AAA
...
# ls -l AAA
total 8
drwxrwxr-x. 2 student student 50 24 fév 11:35 BBB
-rw-rw-r--. 1 student student 569 24 fév 11:34 Compile.sh
-rw-rw-r--. 1 student student 392 24 fév 11:34 TestLSOF.cpp
# ls -l AAA/BBB
total 8
-rw-rw-r--. 1 student student 1174 24 fév 11:35 TestInsert.cpp
-rw-rw-r--. 1 student student 1546 24 fév 11:35 TestSelect.cpp
# tar -cvf AAA.tar AAA
AAA/
AAA/Compile.sh
AAA/TestLSOF.cpp
AAA/BBB/
AAA/BBB/TestInsert.cpp
AAA/BBB/TestSelect.cpp
# ls -l
total 112
drwxrwxr-x. 3 student student 55 24 fév 11:34 AAA
-rw-rw-r--. 1 student student 10240 24 fév 11:36 AAA.tar
...
# tar -tvf AAA.tar
drwxrwxr-x student/student 0 2022-02-24 11:34 AAA/
-rw-rw-r-- student/student 569 2022-02-24 11:34 AAA/Compile.sh
-rw-rw-r-- student/student 392 2022-02-24 11:34 AAA/TestLSOF.cpp
drwxrwxr-x student/student 0 2022-02-24 11:35 AAA/BBB/
-rw-rw-r-- student/student 1174 2022-02-24 11:35 AAA/BBB/TestInsert.cpp
-rw-rw-r-- student/student 1546 2022-02-24 11:35 AAA/BBB/TestSelect.cpp
#
```

Extraction d'une archive

L'exemple suivant montre comment désarchiver le fichier AAA.tar et récréer automatiquement toutes l'arborescence qu'il contient :

```
# rm -r AAA
# ls -l
total 112
-rw-rw-r--. 1 student student 10240 24 fév 11:36 AAA.tar
...
# tar -xvf AAA.tar
AAA/
AAA/Compile.sh
AAA/TestLSOF.cpp
AAA/BBB/
AAA/BBB/TestInsert.cpp
AAA/BBB/TestSelect.cpp
# ls -l
total 112
drwxrwxr-x. 3 student student 55 24 fév 11:34 AAA
-rw-rw-r--. 1 student student 10240 24 fév 11:36 AAA.tar
...
# ls -l AAA
total 8
drwxrwxr-x. 2 student student 50 24 fév 11:35 BBB
-rw-rw-r--. 1 student student 569 24 fév 11:34 Compile.sh
-rw-rw-r--. 1 student student 392 24 fév 11:34 TestLSOF.cpp
# ls -l AAA/BBB
total 8
-rw-rw-r--. 1 student student 1174 24 fév 11:35 TestInsert.cpp
-rw-rw-r--. 1 student student 1546 24 fév 11:35 TestSelect.cpp
#
```

9.2 L'API C pour MySql

Le but de cette section n'est pas d'apprendre le langage SQL et encore moins d'apprendre à créer ou à gérer une base de données MySql. Il s'agit ici d'étudier les fonctions C fournies par MySql (ce que l'on appelle l'API - "Application Programming Interface") pour pouvoir accéder à une base de données MySql et y ajouter/modifier/supprimer des données dans une ou plusieurs de ses tables.

9.2.1 Une base de données exemple

Supposons que nous disposions d'un compte utilisateur sur MySql, ainsi qu'une base de données avec les propriétés suivantes :

- Nom d'utilisateur : **Student**
- Mot de passe utilisateur : **PassStudent1_**
- Base de données : **PourStudent**
- Table présente dans la base de données PourStudent : **Personnes**

Une connexion en ligne de commande à la base de données est fournie par

```
# mysql -u Student -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 15
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| PourStudent   |
| information_schema |
+-----+
2 rows in set (0.00 sec)

mysql> use PourStudent;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
| Tables_in_PourStudent |
+-----+
| Personnes           |
+-----+
1 row in set (0.00 sec)

mysql> describe Personnes;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int        | NO   | PRI | NULL    | auto_increment |
| Nom        | varchar(20) | YES  |     | NULL    |                |
| Prenom      | varchar(20) | YES  |     | NULL    |                |
| AnneeNaissance | int       | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

mysql> select * from Personnes;
+-----+-----+-----+
| id | Nom      | Prenom | AnneeNaissance |
+-----+-----+-----+
| 1  | Issier   | Paul   | 2002            |
| 2  | Coptere  | Eli    | 1991            |
| 3  | Tombale  | Pierre | 1983            |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

La table **Personnes** comporte donc 3 tuples et chaque tuple comporte

- un identifiant entier **id** unique (auto-incrémenté) qui fait office de clé primaire,
- une chaîne de 20 caractères **Nom**,
- une chaîne de 20 caractères **Prenom**, et
- un entier **AnneeNaissance**.

9.2.2 Connexion/déconnexion à une base de données

Afin qu'un programme C puisse communiquer avec une base de données MySql, il est tout d'abord nécessaire de se connecter à la base de données. Une connexion à une base de données MySql est représentée par une structure **MYSQL** définie dans le fichier **mysql.h** et dont le contenu peut rester opaque pour le programmeur applicatif. Cette structure doit tout d'abord être initialisée à l'aide de la fonction

```
#include <mysql.h>

MYSQL* mysql_init(MYSQL *mysql);
```

où **mysql** est un pointeur NULL ou l'adresse d'une variable de type **MYSQL**. Dans le cas où **mysql** est un pointeur NULL, la variable est allouée dynamiquement. Dans les deux cas, la structure est initialisée correctement et la valeur de retour de la fonction est le pointeur alloué ou l'adresse de la variable reçue en paramètre.

Dès qu'une structure **MYSQL** a été correctement initialisée, il est alors possible de **se connecter** à une base de donnée à l'aide de la fonction

```
#include <mysql.h>

MYSQL* mysql_real_connect(MYSQL *mysql,
                           const char *host,
                           const char *user,
                           const char *passwd,
                           const char *db,
                           unsigned int port,
                           const char *unix_socket,
                           unsigned long client_flag);
```

où

- **mysql** est un pointeur vers une variable de type **MYSQL** initialisée à l'aide de la fonction **mysql_init**.
- **host** est une chaîne de caractères contenant l'adresse IP de la machine hébergeant la base de données. Si **host** est NULL ou contient "localhost", la connexion se fait sur la machine locale.
- **user** est une chaîne de caractères contenant le nom de l'utilisateur.
- **passwd** est une chaîne de caractères contenant le mot de passe de l'utilisateur.

- **db** est une chaîne de caractères contenant le nom de la base de données sur laquelle on souhaite se connecter.
- **port** est un entier utilisé dans le cas où la connexion se réalise sur une machine distante. Dans ce cas, il s'agit du port de MySql sur la machine considérée. Si on travaille sur la machine locale, cette valeur peut être mise à 0.
- **unix_socket** spécifique de le nom de la socket de connexion avec MySql si on travaille sur la machine locale. En ce qui nous concerne, on laissera ce paramètre à NULL.
- **client_flag** est un flag entier permettant de paramétriser la connexion plus finement. Son étude sort du cadre de cette introduction. On le laissera donc à 0.

Cette fonction établit donc la connexion avec le serveur MySql dont les propriétés sont précisées dans les paramètres de la fonction. Dans ce cas, le retour de la fonction est le pointeur vers la structure MYSQL, c'est-à-dire la valeur mysql passée en paramètre.

Bien sûr, l'appel à cette fonction **peut échouer** et dans ce cas, le **retour** de la fonction est **NULL**. Il est alors possible de connaître les raisons de cet échec à l'aide de la fonction

```
#include <mysql.h>

const char* mysql_error(MYSQL *mysql);
```

qui retourne une **chaîne de caractères** contenant le **message d'erreur** correspondant au dernier appel de fonction qui a échoué sur la connexion représentée par la variable pointée par **mysql**.

Finalement, la **déconnexion** de la base de données se réalise grâce à la fonction

```
#include <mysql.h>

void mysql_close(MYSQL *mysql);
```

Dans le cas où mysql pointe vers une structure qui a été allouée dynamiquement par la fonction mysql_init, la zone de mémoire correspondante est également libérée.

Remarque importante pour la compilation

Afin de réaliser correctement la compilation d'un programme utilisant les fonctions de la librairie mysql.h, il est nécessaire d'ajouter un certain nombre d'**options de compilation**. Celles-ci peuvent être connues en appelant les commandes suivantes :

```
# mysql_config --cflags
-I/usr/include/mysql -m64
# mysql_config --libs
-L/usr/lib64/mysql -lmysqlclient -lpthread -lm -lrt -lssl -lcrypto -ldl -lresolv
#
```

Il suffit alors de les rajouter sur la ligne de compilation.

Exemple 9.1 (mysql_init, mysql_real_connect, mysql_error, mysql_close). Soit le programme suivant :

```
***** TestConnexion.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <mysql.h>

int main(int argc, char* argv[])
{
    if (argc != 4)
    {
        printf("Trop peu ou trop de paramètres !\n");
        printf("Usage: TestConnection login password NomBD\n");
        exit(1);
    }

    MYSQL* connexion;

    connexion = mysql_init(NULL);

    // Connexion à la base de données
    if (mysql_real_connect(connexion, "localhost", argv[1], argv[2], argv[3], 0, NULL, 0) == NULL)
    {
        fprintf(stderr, "Erreur de connexion à la BD: %s\n", mysql_error(connexion));
        exit(1);
    }

    printf("Connexion établie avec succès à la BD.\n");

    // Déconnexion de la base de données
    mysql_close(connexion);
    exit(0);
}
```

dont quelques exemples d'exécution fournissent

```
# g++ -o TestConnexion TestConnexion.cpp -I/usr/include/mysql -m64
-L/usr/lib64/mysql -lmysqlclient -lpthread -lm -lrt -lssl -lcrypto -ldl -lresolv
# ./TestConnexion Student PassStudent1_ PourStudent
Connexion établie avec succès à la BD.
# ./TestConnexion XXX PassStudent1_ PourStudent
Erreur de connexion à la BD: Access denied for user 'XXX'@'localhost' (using
password: YES)
# ./TestConnexion Student XXX PourStudent
Erreur de connexion à la BD: Access denied for user 'Student'@'localhost' (using
password: YES)
# ./TestConnexion Student PassStudent1_ XXX
Erreur de connexion à la BD: Access denied for user 'Student'@'localhost' to
database 'XXX'
#
```

On constate que

1. Les **options de compilation** ont été ajoutées lors de la création de l'exécutable **TestConnexion**.
2. La variable **connexion** étant un pointeur non alloué, l'appel de la fonction **mysql_init** a reçu en paramètre **NULL** afin qu'elle réalise elle-même l'allocation dynamique d'une

variable de type MYSQL. Le retour de la fonction correspond à l'adresse de la variable allouée, valeur qui est stockée dans la variable connexion.

3. La **première exécution** de TestConnexion a **réussi** étant donné que tous les paramètres étaient corrects.
4. La **seconde** et **troisième** exécutions ont **échoué** suite à de **mauvais identifiants** de l'utilisateur, comme le montre le message fourni par la fonction mysql_error.
5. La dernière exécution a **échoué** car il n'y existe aucune **base de données** appelée XXX pour l'utilisateur Student.



9.2.3 Requêtes d'insertion, de modification et de suppression

L'insertion d'un tuple dans une table de la base de données nécessite la connaissance du langage SQL et ne sera donc pas expliqué en détails ici. De toute façon, et par exemple, il est juste nécessaire de savoir que pour insérer la personne de nom "Wagner", prénom "Jean-Marc" et année de naissance 1974 dans la table Personnes, la requête SQL correspondante est

```
mysql> insert into Personnes values(NULL, 'Wagner', 'Jean-Marc', 1974);
Query OK, 1 row affected (0.01 sec)

mysql> select * from Personnes;
+----+-----+-----+-----+
| id | Nom   | Prenom | AnneeNaissance |
+----+-----+-----+-----+
| 1  | Issier | Paul   | 2002    |
| 2  | Coptere | Eli    | 1991    |
| 3  | Tombale | Pierre | 1983    |
| 4  | Wagner  | Jean-Marc | 1974    |
+----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Au niveau programmation C, il est possible d'exécuter une telle requête en utilisant la fonction

```
#include <mysql.h>

int mysql_query(MYSQL *mysql, const char *stmt_str);
```

où

- **mysql** est le pointeur vers la structure MYSQL représentant la connexion à la base de données sur laquelle on veut exécuter la requête.
- **stmt_str** est une chaîne de caractères contenant la requête SQL.

Si la requête s'effectue correctement, le retour de la fonction est 0, sinon le retour sera non nul.

Exemple 9.2 (mysql_query). Soit le programme suivant :

```
***** TestInsert.cpp *****
```

```

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

int main(int argc,char* argv[])
{
    if (argc != 4)
    {
        printf("Trop peu ou trop de paramètres !\n");
        printf("Usage: TestInsert nom prenom anneeNaissance\n");
        exit(1);
    }

    MYSQL* connexion;
    connexion = mysql_init(NULL);

    // Connexion à la base de données
    if (mysql_real_connect(connexion,
                           "localhost",
                           "Student",
                           "PassStudent1_",
                           "PourStudent",
                           0,NULL,0) == NULL)
    {
        fprintf(stderr, "Erreur de connexion à la BD: %s\n",mysql_error(connexion));
        exit(1);
    }

    printf("Connexion établie avec succès à la BD.\n");

    // Construction et exécution de la requête
    char requete[256];
    sprintf(requete,
            "insert into Personnes values(NULL,'%s','%s',%d);",
            argv[1],
            argv[2],
            atoi(argv[3]));

    if (mysql_query(connexion,requete) != 0)
    {
        fprintf(stderr, "Erreur de mysql_query: %s\n",mysql_error(connexion));
        exit(1);
    }

    printf("Insertion OK.\n");

    // Déconnexion de la base de données
    mysql_close(connexion);
    exit(0);
}

```

dont quelques exemples d'exécution fournissent

```

# ./TestInsert Quettier Patrick 1973
Connexion établie avec succès à la BD.
Insertion OK.
# ./TestInsert Stere Pepin 1983
Connexion établie avec succès à la BD.
Insertion OK.
#

```

tandis qu'en base de données, nous avons

```
mysql> select * from Personnes;
+----+-----+-----+-----+
| id | Nom      | Prenom    | AnneeNaissance |
+----+-----+-----+-----+
| 1  | Issier   | Paul      | 2002        |
| 2  | Coptere  | Eli       | 1991        |
| 3  | Tombale  | Pierre    | 1983        |
| 4  | Wagner   | Jean-Marc | 1974        |
| 5  | Quettier | Patrick   | 1973        |
| 6  | Stere    | Pepin     | 1983        |
+----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>
```

On constate que

1. Afin de ne pas surcharger le nombre d'arguments de l'exécutable **TestInsert**, les identifiants de l'utilisateur et de la base de données ont été codés "en dur" dans l'appel de la fonction **mysql_real_connect**.
2. La construction de la requête SQL a été réalisée dans le buffer **requete** avec la fonction **sprintf**.



Au niveau programmation C, les requêtes de **modification** et de **suppression** se réalisent de manière tout à fait similaire, en formattant une chaîne de caractères à l'aide de la fonction **sprintf** et avec la même fonction **mysql_query**. Il suffit de se rappeler les requêtes SQL correspondantes. Par exemple,

```
mysql> update Personnes set anneeNaissance=1900 where nom='Issier';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> delete from Personnes where nom='Stere';
Query OK, 1 row affected (0.00 sec)

mysql> select * from Personnes;
+----+-----+-----+
| id | Nom      | Prenom    | AnneeNaissance |
+----+-----+-----+
| 1  | Issier   | Paul      |      1900      |
| 2  | Coptere  | Eli       |      1991      |
| 3  | Tombale  | Pierre    |      1983      |
| 4  | Wagner   | Jean-Marc |      1974      |
| 5  | Quettier | Patrick   |      1973      |
+----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

9.2.4 Requête de sélection

Une requête de sélection nécessite la récupération des résultats de la requête au sein du programme C. Ces données “résultats” portent le nom anglais de “**Result Set**” et sont représentées dans la librairie mysql.h par une variable dont le type est **MYSQL_RES**. Il s’agit d’une espèce de tableau dont le nombre de lignes correspond au nombre de tuples trouvés et dont le nombre de colonnes dépend de la requête SQL.

Pour réaliser une requête de sélection, on utilise toujours la fonction **mysql_query** et le résultat de la requête est alors disponible par l’intermédiaire de la variable **MYSQL** représentant la connexion. Afin de récupérer le résultat de la requête dans une variable de type **MYSQL_RES**, il faut utiliser la fonction

```
#include <mysql.h>

MYSQL_RES* mysql_store_result(MYSQL *mysql);
```

où **mysql** est le pointeur vers la structure **MYSQL** représentant la connexion à la base de données sur laquelle on vient de faire une requête de sélection. Le retour de cette fonction est non NULL si la requête a retourné un Result Set. Sinon, soit la requête n'a retourné aucun Result Set, soit une erreur est survenue.

Une fois le Result Set récupéré, il est possible de connaître le **nombre de colonnes** de ce Result Set, il suffit d'utiliser la fonction

```
#include <mysql.h>

unsigned int mysql_num_fields(MYSQL_RES *result);
```

où **result** est un pointeur vers le Result Set récupéré par la fonction `mysql_store_result`.

Il est également possible de **récupérer chaque ligne** du Result Set en utilisant la fonction

```
#include <mysql.h>

MYSQL_ROW mysql_fetch_row(MYSQL_RES *result);
```

où

- **result** est un pointeur vers le Result Set récupéré par la fonction `mysql_store_result`.

- **MYSQL_ROW** est un type de donnée représentant une **ligne entière** du Result Set. En supposant que l'on récupère le retour de cette fonction dans une variable appelée **ligne** (de type **MYSQL_ROW** donc), chaque champ de cette ligne peut s'obtenir en utilisant l'**opérateur []** : `ligne[0]`, `ligne[1]`, ... `ligne[N-1]` où N est le nombre de champs du Result Set. Chaque **champ** d'un Result Set doit alors être interprété comme une **chaîne de caractères**.

Il est à noter qu'il faut appeler la fonction `mysql_fetch_row` autant de fois qu'il y a de lignes dans le Result Set. Cette fonction retourne **NULL** lorsque l'on a atteint la **fin du Result Set**.

Exemple 9.3 (mysql_store_result, mysql_num_fields, mysql_fetch_row). Soit le programme suivant :

```
***** TestSelect.cpp *****/
#include <stdio.h>
#include <stdlib.h>
#include <mysql.h>

int main(int argc,char* argv[])
{
    if (argc != 2)
    {
        printf("Trop peu ou trop de paramètres !\n");
        printf("Usage: TestSelect table\n");
        exit(1);
    }

    MYSQL* connexion;
    connexion = mysql_init(NULL);

    // Connexion à la base de données
    if (mysql_real_connect(connexion,
                          "localhost",
                          "Student",
                          "PassStudent1_",
                          "PourStudent",
                          0,NULL,0) == NULL)
    {
        fprintf(stderr, "Erreur de connexion à la BD: %s\n",mysql_error(connexion));
        exit(1);
```

```

}

printf("Connexion établie avec succès à la BD.\n");

// Construction et exécution de la requête
char requete[256];
sprintf(requete,"select * from %s;",argv[1]);
if (mysql_query(connexion,requete) != 0)
{
    fprintf(stderr, "Erreur de mysql_query: %s\n",mysql_error(connexion));
    exit(1);
}

printf("Requête SELECT réussie.\n");

// Affichage du Result Set
MYSQL_RES *ResultSet;
if ((ResultSet = mysql_store_result(connexion)) == NULL)
{
    fprintf(stderr, "Erreur de mysql_store_result: %s\n",mysql_error(connexion));
    exit(1);
}

MYSQL_ROW ligne;
int nbChamps = mysql_num_fields(ResultSet);
while ((ligne = mysql_fetch_row(ResultSet)) != NULL)
{
    for(int i=0; i<nbChamps ; i++)
        printf("%10s\t",ligne[i]);

    printf("\n");
}

// Déconnexion de la base de données
mysql_close(connexion);
exit(0);
}

```

dont quelques exemples d'exécution fournissent

```

# ./TestSelect Agents
Connexion établie avec succès à la BD.
Erreur de mysql_query: Table 'PourStudent.Agents' doesn't exist
# ./TestSelect Personnes
Connexion établie avec succès à la BD.
Requête SELECT réussie.
      1      Issier      Paul      1900
      2      Coptere     Eli      1991
      3      Tombale     Pierre    1983
      4      Wagner      Jean-Marc 1974
      5      Quettier    Patrick   1973
#

```

On constate que

1. Le premier argument de l'exécutable **TestSelect** correspond au nom de la table dans laquelle on souhaite réaliser une sélection.
2. La **première exécution** de TestSelect a échoué étant donné que la table Agents n'existe

pas dans la base de données. Dans ce cas, c'est la fonction `mysql_query` qui a échoué.

3. La **seconde exécution** a permis d'afficher **tous les tuples** de la table **Personnes**. Le nombre de champs de chaque tuple a été récupéré dans la variable **nbChamps** et celui-ci correspond bien à **4**.
4. Quels que soient les types des champs dans la base de données, les champs sont tous récupérés sous la forme de **chaîne de caractères** (y compris **id** et **AnneeNaissance**) et affichés par `printf` en utilisant `%s`. S'il est nécessaire de récupérer un champ sous la forme d'un `int` par exemple, on devra utiliser la fonction **atoi**.



Index

#define, 21
#endif, 23
#ifdef, 23
#include, 20
+s, 116
-D, 24
-I, 26
-Wall, 26, 27
.SILENT :, 54
/etc/group, 62
/etc/passwd, 61

access, 86, 87
alarm, 201, 202
argc, 39
argument, 13
argv, 39
armer un signal, 155, 157
asynchrone, 109, 160
atomique, 275

boîte aux lettres, 213

chdir, 322, 323
chmod, 64, 85, 86, 116
chown, 73
clé, 206, 210
clean, 57
client, 233
client/serveur, 233
clobber, 58
close, 73, 78, 79, 289
closedir, 326
commande, 13
core, 132, 136
creat, 73

démons, 111, 237
defunct, 171
Descripteur de fichier, 73
droits, 63
droits d'un répertoire, 63
dup, 90, 92
dup2, 90, 94

Edition de liens, 28
ENOMSG, 230
ensemble de sémaphores, 261
errno, 28
EUID, 115
exec, 117, 138
execl, 138, 139
execlp, 138, 142
execv, 138, 141
execvp, 138
exit, 116, 117, 127, 133

F_DUPFD, 310
F_GETFD, 310
F_GETFL, 310
F_GETLK, 310, 317
F_SETFD, 310
F_SETFL, 310
F_SETLK, 310, 312
F_SETLKW, 310, 315
fclose, 70
fcntl, 73, 294, 299, 309, 312, 315, 317
fdopen, 103, 104
fileno, 100, 101
Fin de fichier, 81
fopen, 70, 100
fork, 117, 120, 122, 296
fprintf, 36
fread, 70
fseek, 70
fstat, 95
fwrite, 70

g++, 17
gcc, 17
GETALL, 268
getegid, 115
geteuid, 115
getgid, 115
GETNCNT, 268
getpgrp, 188
GETPID, 268
getpid, 113

getppid, 113
getuid, 115
GETVAL, 268
GETZCNT, 268
grep, 302

handler de signal, 150, 157
HOME, 108

i-noeud, 74
id, 62, 116
identifiant, 215
identificateur, 215
IPC, 205
IPC_NOWAIT, 224, 227, 230, 276
ipc_perm, 206
IPC_RMID, 220, 244, 268
IPC_SET, 220, 244, 268
IPC_STAT, 220, 244, 268
ipcrm, 209, 210
ipcs, 207

key_t, 206
kill, 150, 151, 194–196, 199

lecture destructrice, 227
link, 73
lseek, 73, 82, 83
lsof, 331

Macro paramétrée, 22
make, 49
makefile, 49
masquer un signal, 155, 174
message, 222
mkdir, 319–321
mode noyau, 112
mode utilisateur, 112
MSG_EXCEPT, 230
MSG_NOERROR, 230
msgctl, 219, 221
msgflg, 230
msgget, 216–218
msgrcv, 226, 227
msgsnd, 223, 224
MySql, 337
mysql_close, 340, 341
mysql_error, 340, 341
mysql_fetch_row, 346
mysql_init, 339, 341
mysql_num_fields, 345, 346
mysql_query, 342

mysql_real_connect, 339, 341
mysql_store_result, 345, 346

noyau Linux, 9

opération atomique, 259
open, 73, 76, 79
opendir, 326
option, 13
ordonnanceur, 111
orphelin, 121

paramètre, 13
PATH, 108
pause, 198
perror, 32
PID, 107, 113
pipe, 83, 288, 289, 291, 294
PPID, 108, 113
Problèmes de buffer, 70
Process Identifier, 107
processus, 107
programme, 107
propriétaire effectif, 114
propriétaire réel, 114

réentrant, 112
read, 73, 80, 291, 294
readdir, 326
redirection, 36, 37, 102
Result Set, 345
rewinddir, 328
rmdir, 321

sa_flags, 165
sa_mask, 162
SA_NODEFER, 165, 167
SA_RESETHAND, 165
SA_RESTART, 165, 166
savesigs, 186
scheduler, 111
section critique, 176, 177
seekdir, 328
SEM_UNDO, 276
semctl, 261, 268, 270, 273
semget, 261, 263, 265, 267
semop, 261, 275, 277
serveur, 233
serveur mono-processus, 237
serveur multi-processus, 237
session, 189
SETALL, 268

- setpgid, 189, 190, 192
SETVAL, 268
shell, 108
shmat, 247, 248, 251
shmctl, 244, 245
shmdt, 253, 254
shmget, 239, 241, 242
SIG_BLOCK, 174
SIG_SETMASK, 174
SIG_UNBLOCK, 174
sigaction, 158, 160
sigaddset, 155
SIGALRM, 201
SIGCHLD, 118, 170, 172
sigdelset, 155
sigemptyset, 155
sigfillset, 155
SIGFPE, 185
sigismember, 155
siglongjmp, 181, 182
signal bloqué, 153
signal délivré, 153
signal ignoré, 153
signal masqué, 153
signal pendant, 153
sigpending, 178, 179
sigprocmask, 174, 175
sigset_t, 155, 156
sigsetjmp, 181, 183
sleep, 199
snprintf, 35
sprintf, 35
st_mode, 97
stat, 73, 95, 97
stdin, 126, 301
stdout, 301
strerror, 31
struct msg, 212
struct msqid_ds, 211
struct sembuf, 275
struct semid_ds, 261
struct shmid_ds, 238
struct sigaction, 159
su, 116
substitution d'identité, 116
synchrone, 109
- Table des fichiers ouverts, 74, 288
Table des files de messages, 211
Table des mémoires partagées, 238
Table des sémaphores, 261
- Tables systèmes, 73
tar, 336
telldir, 328
temporisation, 201
time out, 201
touch, 54
Trace, 38
tube, 287
tube anonyme, 287
type d'un message, 214, 223
- UID, 114
umask, 88, 89
union semun, 268
unlink, 73
- wait, 117, 129, 130, 133, 134, 136
WCOREDUMP, 132, 136
WEXITSTATUS, 132
WIFEXITED, 132
WIFSIGNALED, 132, 134, 136
write, 73, 80, 291
WTERMSIG, 132, 134, 136
- zombi, 113, 170, 172

Références

1. Système d'exploitation UNIX. DENYS MERCENIER. 2017. Haute Ecole de la Province de Liège.
2. Développement système sous Linux - Ordonnancement multitaâche, gestion mémoire, communications, programmation réseau. 5ème édition. CHRISTOPHE BLAES. 2019. Editions Eyrolles. ISBN : 978-2-212-67760-7.
3. Les systèmes d'exploitation - Unix, Linux, Windows XP avec C et Java. SAMIA BOUZE-FRANE. 2003. Editions Dunod. ISBN : 2-10-007189-0.
4. Wikipedia : <https://fr.wikipedia.org/wiki/Linux>.