Université Mohammed Premier
Faculté Pluridisciplinaire
Département de Mathématiques et Informatique
◇ Nador ◇

Programmation Système en C sous Linux

◇ **Filière**: SMI, S6 ◇

A.U : 2019-2020

Pr. K. El Makkaoui
kh.elmakkaoui@gmail.com

Février 2020

Sommaire

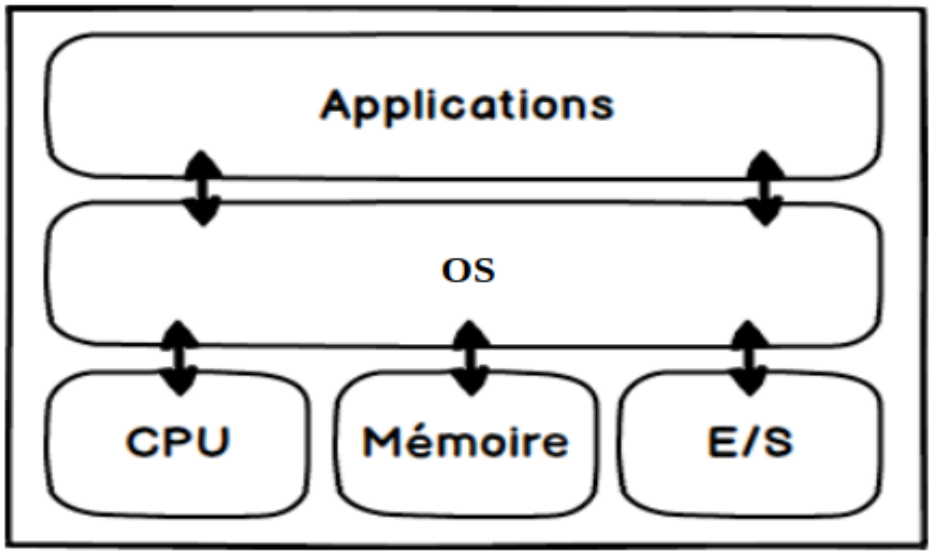
L'objectif de ce chapitre est de:

▷ présenter les principes généraux de la programmation sous Linux.

▷ présenter les outils disponibles pour réaliser des applications et les étapes de base nécessaires à la création des programmes Linux en langage C.

▷ expliquer comment créer, modifier et compiler un code source C et déboguer le résultat.

Un système d'exploitation "SE" (operating system en anglais) est un logiciel spécialisé qui sert principalement à gérer le lien entre les ressources matérielles d'un ordinateur (RAM, CPU et périphériques E/S) et les applications informatiques de l'utilisateur.



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

▷ Les **programmes d'application** des utilisateurs. Ces sont réalisés lors de la programmation dite "classique".
`Exemple:` les applications développées en langage C.

▷ Les **programmes systèmes** qui permettent le fonctionnement de la machine (ordinateur, tablette, raspberry pi, ...).
`Exemples:` L'accès aux fichiers, la gestion des processus, les entrées/sorties, la gestion de la mémoire, la programmation réseau.

N.B. le langage C a été créé spécialement pour la programmation système, plus précisément pour le développement de systèmes d'exploitation **Unix**.

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

La base du système d'exploitation est le **noyau**, qui est une sorte logiciel d'arrière-plan qui assure les communications entre les programmes (inclus les programme de système et programmes d'application). C'est donc par lui qu'il va nous falloir passer pour avoir accès aux informations du système.

Les composants principaux du noyau Linux: le composant d'E/S, le composant de gestion de la mémoire et composant de gestion des processus.

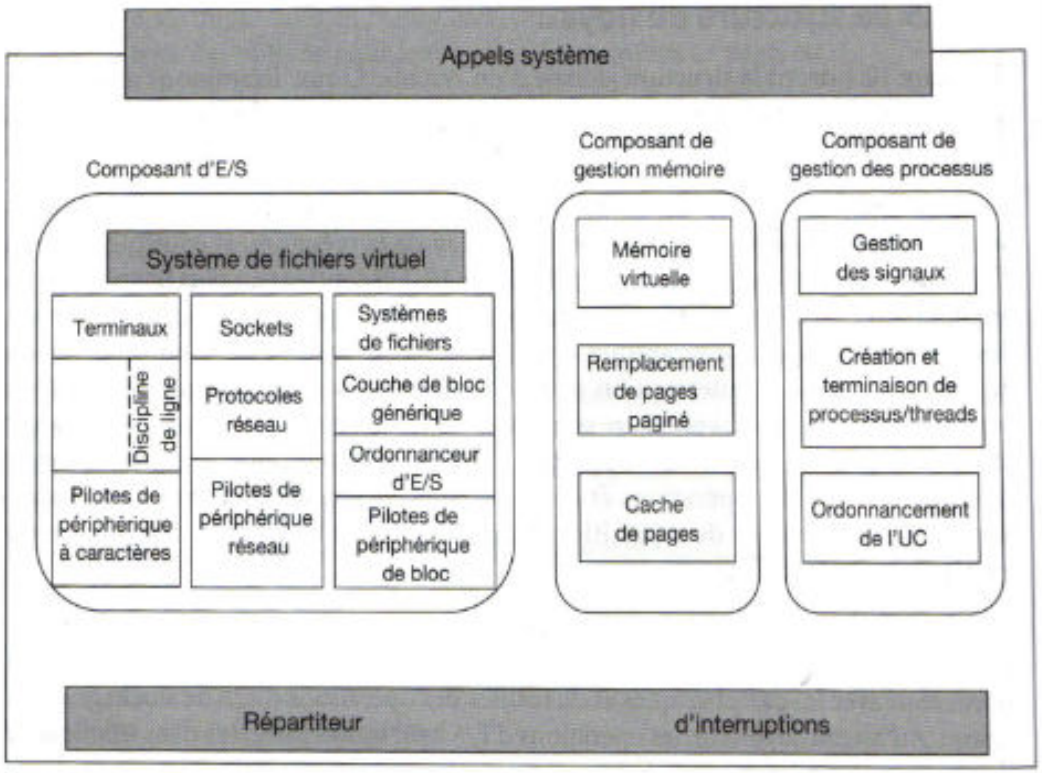


Figure: Structure du noyau Linux

Le noyau fournit des points d'entrées (des fonctions appelées les **appels-systèmes**) pour accéder à ces informations.

L'appel système est l'interface fondamentale entre une application et le noyau Linux.

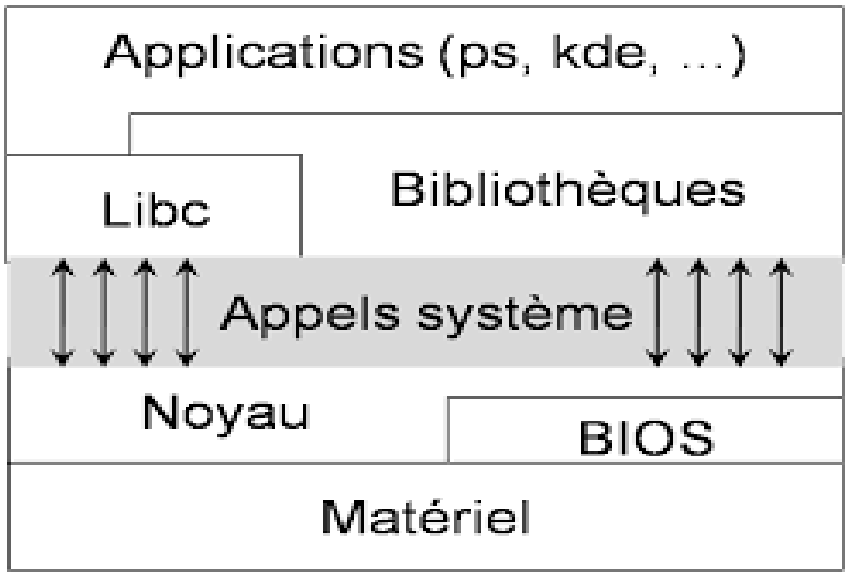


Figure: Schéma d'un système d'exploitation de type Linux

Le développement en C sous Linux comme sous la plupart des autres SEs met en œuvre principalement cinq types d'utilitaires :

- `L'éditeur de texte.`
- `Le compilateur.`
- `L'éditeur de liens.`
- `Le débogueur.`
- `D'utilitaires annexes:` travaillant à partir du code source, comme les outils de documentation automatique, etc.

L'`éditeur de texte`, qui est à l'origine de tout le processus de développement applicatif. Il nous permet de créer et de modifier les fichiers source.

Chaque programmeur a généralement son éditeur préféré, dans la suite de ce cours nous utiliserons deux éditeurs de texte à savoir **Vi** et **Gedit**.

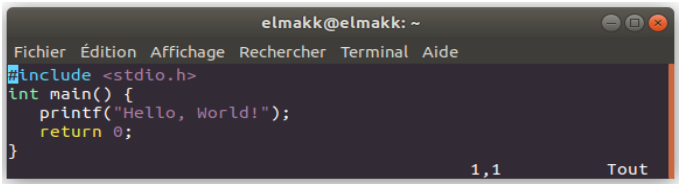


Figure: Vi sous Ubuntu

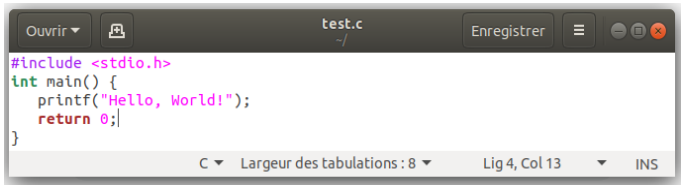


Figure: Gedit sous Ubuntu

Les avantages de l'éditeur de texte **Vi** sont sa disponibilité sur toutes les plates-formes Unix et la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de configuration.

Il existe deux modes de travail:

- En **mode commande**: on appuie sur la touche «`i`» ou «`a`» pour passer en mode insertion.

- En **mode insertion**: le retour au **mode commande** étant réalisé à l'aide de la touche <ESC>.

▷ Depuis de Shell, pour appeler **Vi**: **vi nom_fichier**
↪ Crée le fichier (**nom_fichier**) s'il n'existe pas et si on a les droits.

En mode commande:

▷ **:w** : pour sauvegarder un fichier.

▷ **:q!** : pour sortir sans sauvegarder.

▷ **:x** ou **:wq!** : pour sauvegarder et sortir.

▷ **x** : pour effacer un caractère.

▷ **dd** : pour effacer une ligne (**ndd** : pour effacer n lignes).

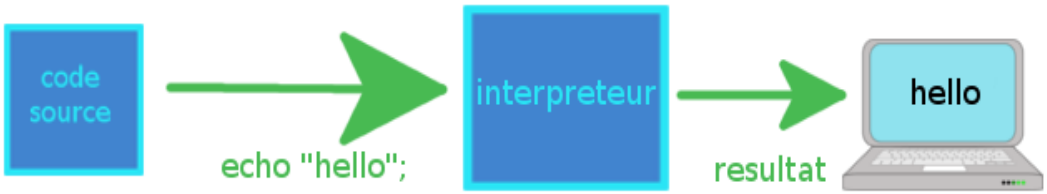
Pour utiliser l'éditeur de texte **Gedit**, on doit d'abord l'installer.

Sous Ubuntu

```
$ sudo apt-get install gedit
```

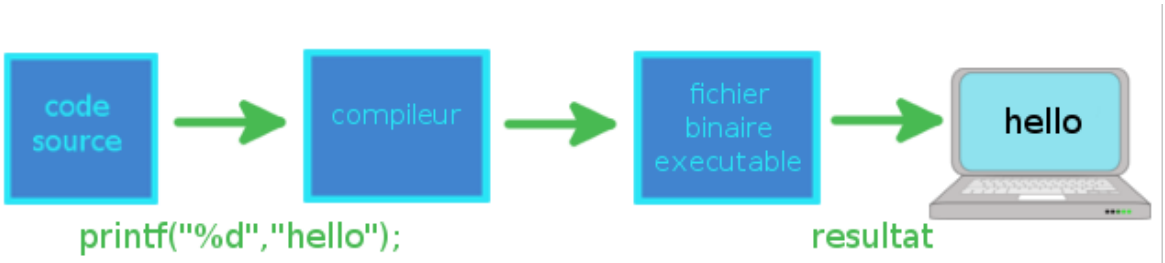
- Depuis de Shell, on peut aussi lancer le **Gedit**: **gedit nom_fichier**

• Un langage **interprété** est un langage dont les implémentations exécutent des instructions directement sans passer par une phase de compilation (script shell, python, PHP, JavaScript, ...)



- Avantages :
  - Multi-plateforme;
  - Simple à tester;
  - Facile à débugger;
  - Code source public.
- Inconvénients :
  - Requiert un interpréteur;
  - Pour chaque exécution, le programme doit être interprété préalablement.

• Un langage **compilé** est un langage qui requiert un compilateur pour traduire le code source en programme binaire compréhensible par la machine. Une fois compilé, le programme peut être démarré et distribué sur la même plateforme (C, C++, ...).



- Avantages :
  - Programme immédiatement disponible à démarrer;
  - Plus rapide, car il est optimisé pour le CPU;
  - Code source privée.
- Inconvénients :
  - Non multi-plateforme;
  - Nécessite des étapes supplémentaires pour tester.

Le langage **C** est un langage compilé.

▷ Le compilateur **C** utilisé sous Linux est **gcc** (`Gnu Compiler Collection`). On peut également l'invoquer sous le nom **cc**, comme c'est l'usage sous Unix, ou **g++** si on compile du code **C++**. Il existe aussi une version nommée **egcs**, il s'agit d'une implémentation améliorée de **gcc**.

▷ Le compilateur **gcc** permet de produire un fichier exécutable à partir d'un programme écrit en langage C.

Le **gcc** effectue les tâches suivantes à partir d'un fichier «.c»

- `le préprocesseur` (the preprocessing): nommé **cpp**, gère toutes les directives (#include, #define, #ifdef, ...) du code source.

- `le compilateur` (the compiling): traduit le code source en code assembleur, un langage très proche du langage machine.

- `l'assembleur` (the assembling): traduit le code assembleur en code (langage) machine. On obtient alors le fichier objet.

- `l'éditeur de liens` (the linking): assure le regroupement des fichiers objet et des bibliothèques pour fournir enfin le fichier exécutable.

Pratiquement, par exemple à partir d'un fichier «programme.c», la compilation consiste à effectuer les étapes suivantes:

- Passer le préprocesseur: générer le fichier «programme.i».
- Générer un fichier assembleur «programme.s».
- Faire l'assemblage de ce fichier pour obtenir «programme.o».
- Faire l'édition de liens avec les bibliothèques utiles.

Installation le **gcc** sous Ubuntu:

Pour installer le compilateur **gcc**, on ouvre un terminal et on tape les commandes suivantes:

```
// passer en mode super-utilisateur:
 $ sudo su
// installer le gcc:
 # apt-get install gcc
```

Pour connaître la version de **gcc** qui installée, on tape la commande suivante:

```
 $ gcc -v ou gcc –version
```

Pour compiler et produire un fichier exécutable on tape la commande **gcc**.

Syntaxe:

```
$ gcc nom_source.c
```

↪ «nom_source.c»: le fichier «.c» à compiler.

Par défaut, la compilation produit en sortie un fichier exécutable nommé «**a.out**». Pour donner un nom explicite au fichier exécutable, on utilise l'option «-o» (out) de **gcc**.

Syntaxe:

```
$ gcc nom_source.c -o nom_sortie
```

Pour exécuter le fichier exécutable on tape la commande:

```
// pour la compilation par défaut:
 $ ./a.out
// pour la compilation avec un nom explicite au fichier exécutable:
 $ ./nom_sortie
```

L'option «**-c**» de la commande «**gcc**» permet de produire un fichier objet du fichier source (c-à-d elle effectue les trois premières étapes de la compilation mais elle ne fait pas d'édition de lien).

Exemple:

```
$ gcc -c programme.c -o programme.o
```

↪ «programme.o» c'est le fichier objet généré. Il contient des informations codées en binaire (format presque exécutable) qui représentent le programme contenu dans le fichier source.

La compilation séparée:

Un gros programme peut être découpé en plusieurs fichiers indépendants. En effet la séparation du code est la moyen d'avoir des parties du code réutilisables (c-à-d, des fonctions qui peuvent servir dans d'autres programmes).

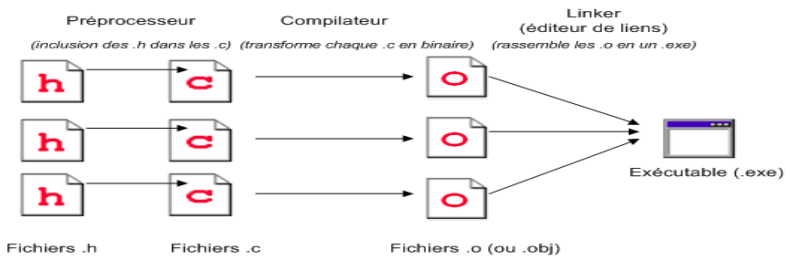
Lorsqu'on compile un programme qui utilise plusieurs fichiers, il faut donner les noms de tous les fichiers à utiliser et un des fichiers doit contenir la fonction principale «`main()`».

Exemple:

Soient `prog1.c`, `prog2.c`, ..., `prog.c` des fichiers sources où «`prog.c`» contient la fonction «main()». Pour compiler et générer l'exécutable:

```
$ gcc prog1.c prog2.c ... prog.c -o prog
```

On peut aussi, tout d'abord générer les fichiers objets ensuite les compiler.



Exemple:

```
$ gcc -c prog1.c -o prog1.o
$ gcc -c prog2.c -o prog2.o
...
$ gcc -c prog.c -o prog.o
$ gcc prog1.o prog2.o ... prog.o -o prog
```

Remarque: au lieu de garder les fichiers sources de ces fonctions qui peuvent servir d'autres programmes, on peut les rassembles dans des bibliothèques.

Quelques options de gcc:

– l'option «-o nom»: donne le nom du fichier de sortie.
– l'option «-c»: s'arrête au fichier «.o».
– l'option «-g»: génére les informations pour le débogueur.
– l'option «-lx»: ajouter la bibliothèque «x» lors de l'édition de liens.
– l'option «-L chemin_repertoire_lib»: recherche librairies dans ce chemin puis dans le chemin standard /lib.
– l'option «-w»: supprime tous les warnings.
– l'option «-W»: active les warnings supplémentaires.
– l'option «-Wall»: active tous les warnings possibles.
– l'option «...»:

Pour que plusieurs fichiers soient utilisables dans un programme principal, il faut donner les prototypes des fonctions définies dans ces fichiers et qui sont utilisées dans le programme. Les prototypes sont écrits dans des fichiers en-tête qui sont caractérisés par l'extension «`.h`».

▷ L'inclusion d'un fichier en-tête est réalisée par les instructions:

```
#include "nom_fichier.h"
ou
#include <nom_fichier.h>
```

– Pour #include "...", la recherche des fichiers «`.h`» s'effectue dans le répertoire courant.
– Pour #include <...>, la recherche des fichiers «`.h`» s'effectue dans les répertoires d'inclusion «`/usr/include`».
– On peut aussi spécifier des sous répertoires des répertoires d'inclusion. Par exemple, #include <sys/types.h>

Le débogueur permet d'exécuter pas à pas le code, d'examiner des variables internes, etc. En autre terme, il aide un développeur à analyser les bugs d'un programme.

Lorsqu'une application a été compilée avec l'option «g», il est possible de l'exécuter sous le contrôle d'un débogueur.

L'outil utilisé sous Linux est nommé **gdb** (Gnu Debugger); cet utilitaire fonctionne en ligne de commande, avec une interface assez rébarbative.

Un autre débogueur est disponible sous Linux, nommé **ddd** (Data Display Debbuger), plus agréable visuellement.

Installation de gdb et ddd:

```
$ sudo apt-get install gdb
$ sudo apt-get install ddd
```

Depuis le Shell, pour lancer le débogueur **ddd**:

```
$ ddd file
```

↪ «file»: le fichier exécutable, généré en utilisant l'option «g» du compilateur «gcc».

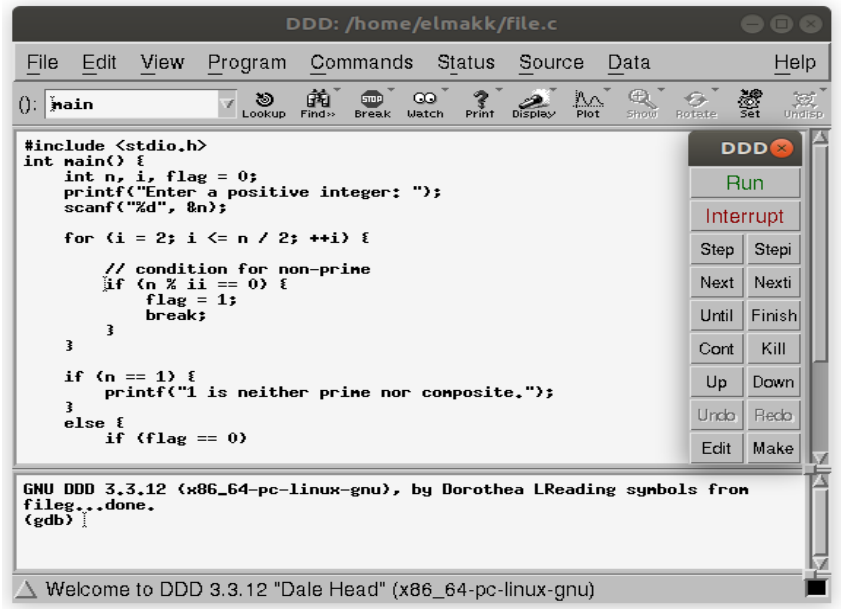


Figure: Utilisation de ddd

L'objectif de ce chapitre est de:

▷ apprendre à maîtriser les processus de bout en bout, de leur création jusqu'à leur terminaison.

▷ apprendre à exécuter un programme externe à partir de notre programme.

▷ présenter les principales commandes d'environnement.

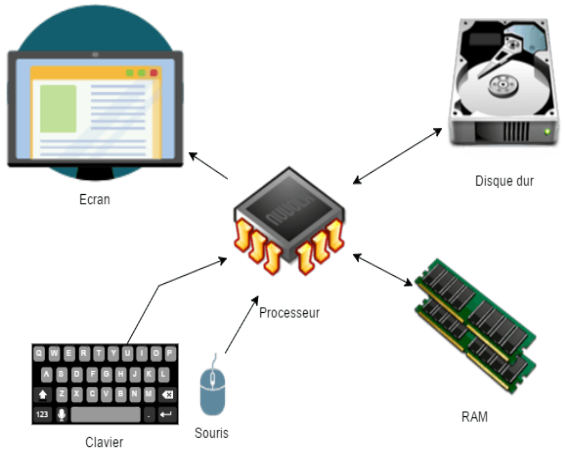
▷ présenter une famille de fonction qui nous permet de lancer des programmes par la famille **exec**.

▷ ...

• Sous Unix, tout programme qui est en cours d'exécution (i.e., programme à l'état actif) est représenté par un processus.

• Chaque processus est identifié par un numéro (Process IDentifier "PID").

• Puisque le processeur ne peut exécuter qu'une seule instruction à la fois, le noyau va donc découper le temps en tranches de quelques millisecondes (`quantum` de temps) et attribuer chaque quantum à un programme (processus).
↪ Réellement le processeur bascule entre les programmes.
↪ Mais l'utilisateur voit ses programmes s'exécuter en même temps (en parallèle).



• Pour lancer un programme, situé dans le disque dur, un (ou plusieurs) **processus** sera créer.

• Chaque processus réserve un espace mémoire pour stocker ses informations et utilise (en pourcentage) le CPU.

• Les processus sont organisés en hiérarchie. Chaque processus (`le processus fils`) doit être lancer par un autre (`le processus père`). La racine de cette hiérarchie est le programme initial (nommé **init**, “PID=1” et n’a pas de père “PPID=0”).

On peut examiner la liste des processus en cours sur le système Unix à l'aide des commandes suivantes:

```
$ ps -ef
$ ps -aux
$ ps -axj
$ top
```

On peut afficher les processus sous forme d'arborescence et les visualiser par leurs liens de parenté, en utilisant la commande suivante.

```
$ pstree
```

Pour implémenter les processus, le système d'exploitation utilise un tableau de structure (appelé **table des processus**). Ce table comprend une entrée par processus, allouée dynamiquement, c'est le **bloc de contrôle du processus** (Process control block "PCB"). Ce bloc contient entres autres, les informations suivantes:

- Le `PID`, le `PPID`, l'`UID` et le `GID` du processus.
- L'état du processus.
- Les fichiers ouverts par le processus.
- Le répertoire courant du processus.
- Le terminal attaché au processus.
- Les signaux reçus par le processus.
- Le contexte processeur et mémoire du processus.

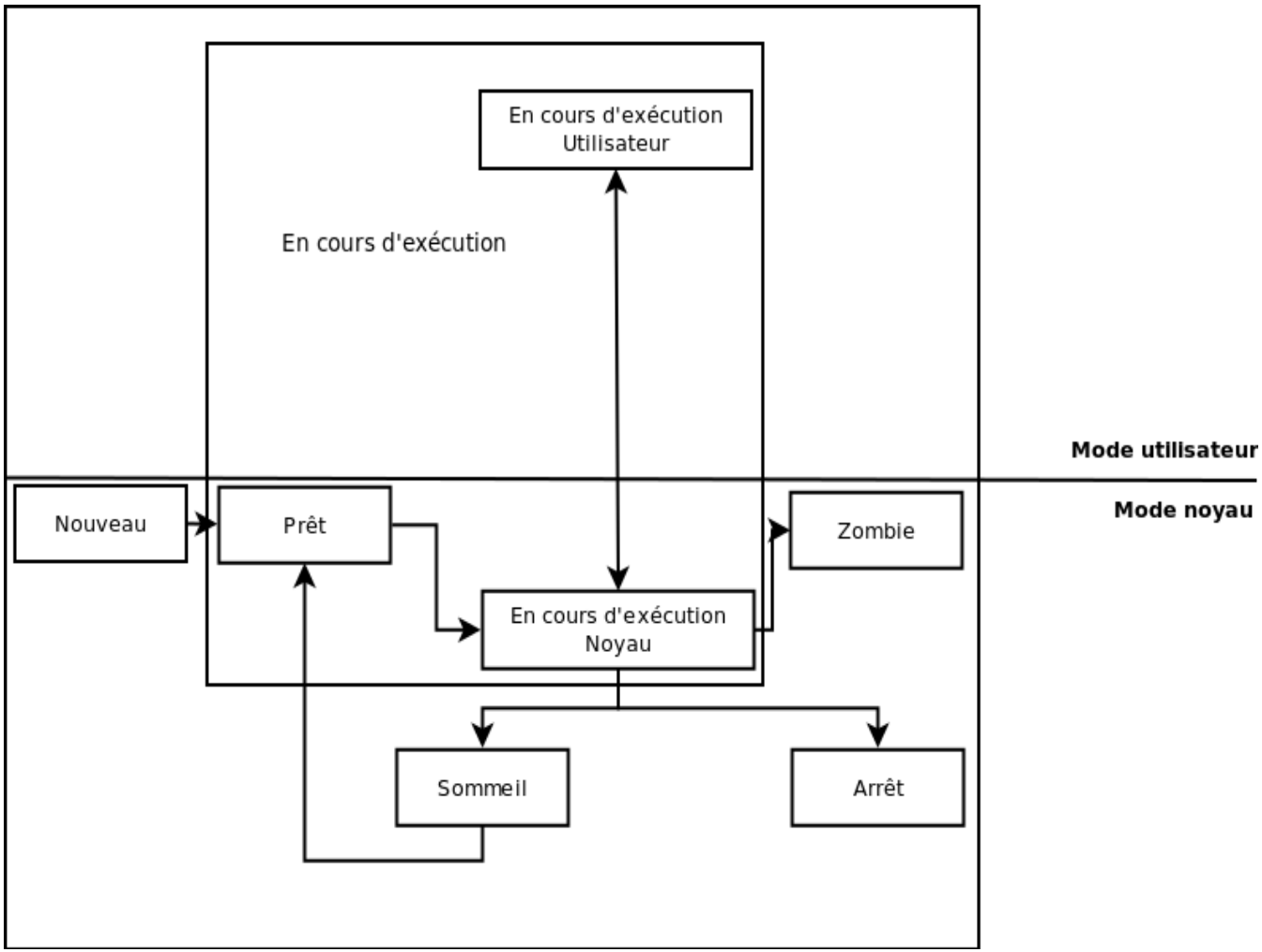
↪ Grâce à ces informations stockées dans la table des processus, un processus bloqué ou arrêté pourra redémarrer ultérieurement avec les mêmes caractéristiques.

Un processus peut avoir plusieurs états:

- **Exécution** (running "**R**"): le processus est en cours d'exécution.
- **Sommeil** (sleeping "**S**"): le processus est en attente d'accomplissement d'un événement.
- **Arrêt** (stopped "**T**"): le processus a été temporairement arrêté par un signal; en utilisant par exemple: `kill -19 PID`.
- **Zombie** ("**Z**"): le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

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

Un diagramme d'états des processus sous Unix:



Pour connaître l'identifiant "PID" du processus appelant, qui lui attribué par le système au moment de sa création, on utilise l'appel-système (la fonction) `getpid()`.

L'appel-système `getpid()` est déclaré dans la bibliothèque **<unistd.h>**, qui ne prend pas d'argument et renvoie une valeur de type `pid_t` (il s'agit généralement d'un `int` et il est déclaré dans **<sys/types.h>**).

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

L'appel système `getppid` retourne le `PPID` du processus appelant:

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

L'appel système `getuid` retourne l'`UID` du processus appelant:

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
```

L'appel système `getgid` retourne le `GID` du processus appelant:

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

Exercice 1: Créer un programme en C qui permet d'afficher `PID`, `PPID`, `UID` et `GID` du processus appelant.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
int main(void)    {
   system("ps -f");
   printf ("%d PID", getpid());
   printf ("%ld PPID", (long) getppid());
   printf ("%d UID", getuid());
   printf ("%d GID", getgid());
   return 0;    }
```

N.B. le type de `pid_t` est un entier sur 64 bits, mais n'est pas le cas pour tous les Unix. Pour assurer la portabilité lors de l'affichage d'un PID, nous utiliserons la conversion `%ld` de `printf()`, et nous ferons explicitement une conversion de type en `long int`.

Le premier processus du système, `init`, ainsi que quelques autres sont créés directement par le noyau au démarrage.

On peut créer un nouveau processus en utilisant l'appel-système `fork()`.

`fork()` va dupliquer le processus appelant. Au retour de cet appel-système, deux processus identiques continueront d'exécuter le code à la suite de `fork()`.

La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original (le processus père) et la nouvelle copie (le processus fils).

L'appel système `fork()` est déclaré dans la bibliothèque **<unistd.h>**, ainsi:

```
pid_t fork(void);
```

Les processus père et fils ne se distinguent que par la valeur de retour de `fork()`:

- Dans le processus père: l'appel-système `fork()` renvoie le numéro du processus fils.
- Dans le processus fils: l'appel-système `fork()` renvoie **0**.
- En cas d'échec, le processus fils n'est pas crée, `fork()` renvoie **-1**.

Examiner l'exécution du programme suivant:

```
#include <unistd.h>
#include <stdlib.h>
int main(void)
{
   fork();
   system("ps -f");
   return 0;
}
```

Les deux processus, généralement de même priorité, sont exécutés selon la politique du tourniquet (Round Robin).

Exercice 2: Exploiter les valeurs de retour de la fonction `fork()` pour exécuter les deux processus l'un après l'autre (objectif est de filtrer les sorties de père et de fils).

```c
#include<stdio.h>
#include<unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void){
pid_t id;
id = fork();
switch (id)
{
        case -1: printf("Erreur!"); break;
        case 0:
        printf("Fils\n");
        system("ps -f");
        break;
        default:
        wait(NULL);
        printf("Père\n");
        system("ps -f");
        break;
}
return 0;
}
```

Dans le cas où la fonction `fork()` renvoie «**-1**» c'est qu'il a eu une erreur, le code de cette erreur est placé dans la variable globale «**errno**», déclaré dans le fichier d'en-tête «**errno.h**».

La fonction «**perror()**» renvoie une chaîne de caractère décrivant l'erreur dont le code est stocké dans la variable «**errno**».

Syntaxe

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

- **perror ()** imprime un message d'erreur descriptif sur stderr.
- **str**: il s'agit de la chaîne C contenant un message personnalisé à imprimer avant le message d'erreur.

Ce code peut correspondre à deux constantes:

• **ENOMEM**: le noyau n'a plus assez de mémoire disponibles pour créer un nouveau processus.

• **EAGAIN**: ce code d'erreur peut être dû a deux raisons: soit il n'y a pas suffisamment de ressources systèmes pour créer le processus, soit l'utilisateur a déjà trop de processus en cours d'exécution.

Un programme peut se finir de plusieurs manières. La plus simple est de laisser le processus finir l'exécution de la fonction **main** avec l'instruction **return** suivie de la valeur de retour du programme. Cette valeur est lue par le processus père, qui peut en tirer les conséquences adéquates. Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échec sont indiqués par des codes de retour non nuls.

Une autre façon de terminer un programme normalement est d'utiliser la fonction `exit()`.

L'appel système `exit()` est déclaré dans la bibliothèque **<stdlib.h>**:

```
void exit (int code);
```

Exemple:

```c
#include <stdio.h>
#include <stdlib.h>
void quit(void)
{
  printf("Nous sommes de la fonction quit() \n");
  exit (0);
}

int main (void){
quit();
printf("Nous sommes de la fonction main() \n");
return 0;
}
```

Un programme peut également se terminer de manière anormale. Ceci est le cas par exemple lorsqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé. Ces actions déclenchent un signal qui, par défaut, arrête le processus.

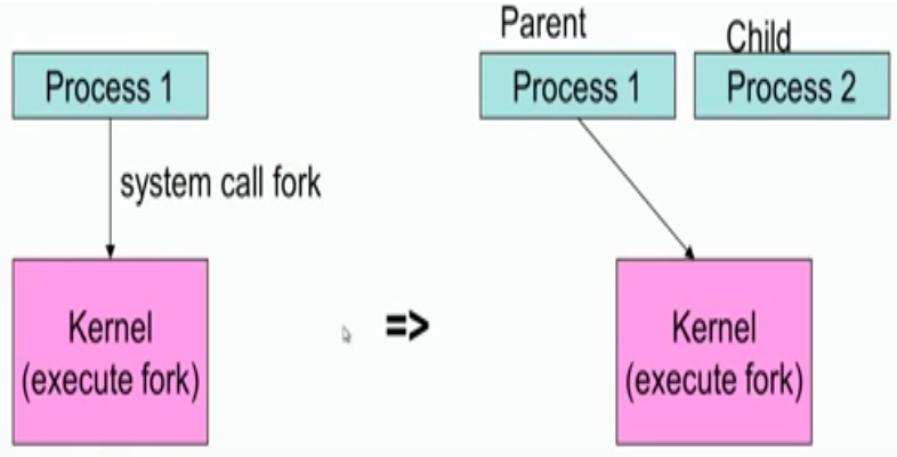
Une manière « propre » d'interrompre anormalement un programme (par exemple lorsqu'un bogue est découvert) est d'invoquer la fonction `abort()`.

```
void abort (void);
```

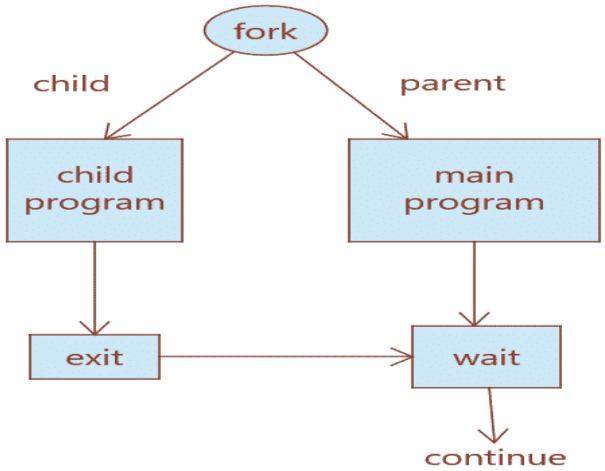
↪ Celle-ci envoie immédiatement au processus le signal SIGABRT.

Création d'un processus par clonage:

- Le processus fils est une réplique exacte du processus père.
- Appel système `fork()`.



Lorsqu'on invoque l'appel système `wait()`, il bloque le processus appelant jusqu'à ce qu'un de ses processus fils se termine.



L'appel système `wait()` est déclaré dans la bibliothèque <sys/wait.h>:

```
pid_t wait(int *statut);
```

• Si le pointeur `statut` est non `NULL`, il est renseigné avec une valeur informant sur les circonstances de la mort du processus fils. Si on n'est pas intéressé par les circonstances de la fin du fils, il est tout à fait possible de fournir un argument `NULL`.

• L'appel système `wait()` a le comportement suivant:
– Si le processus appelant n'a pas de fils, il retourne -1, en plaçant l'erreur `ECHILD` dans `errno`.
– Si le processus appelant a au moins un fils `zombie`, le `zombie` est détruit et son `pid` est retourné par `wait()`.
– Si aucun des fils du processus appelant n'est un `zombie`, `wait()` bloque en attendant la mort d'un fils.

La manière dont sont organisées les informations au sein de l'entier `statut` est opaque, et il faut utiliser les macros suivantes pour analyser les circonstances de la fin du fils :

• **WIFEXITED**(`statut`) est vraie si le processus s'est terminé de façon normale en invoquant `exit()` ou en revenant de `main()`. On peut obtenir le code de retour du processus fils, c'est-à-dire la valeur transmise à `exit()`, en appelant **WEXITSTATUS**(`statut`).

• **WIFSIGNALED**(`statut`) indique que le fils s'est terminé à cause d'un signal, y compris le signal `SIGABRT`, envoyé lorsqu'il appelle `abort()`. Le numéro du signal ayant tué le processus fils est disponible en utilisant la macro **WTERMSIG**(`statut`).

• **WIFSTOPPED**(`statut`) indique que le fils est stoppé temporairement. Le numéro du signal ayant stoppé le processus fils est accessible en utilisant **WSTOPSIG**(`statut`).

Exemple:

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

int main(){
pid_t pid;
int statut;
int x=23;
int y= 19;
if (fork()==0) //fils
{
        printf("Je suis le fils, mon PID est: %d \n", getpid());
        return (x+y);
}else //père
{
        pid = wait (&statut);
}
printf("PID de père = %d \n", getpid());
printf("PID de fils = %d , return statut : %d \n", pid, WEXITSTATUS(statut));
return 0;
}
```

L'appel système `waitpid()` est une version étendue de `wait`.

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

• Le paramètre `pid` indique le processus à attendre; lorsqu'il vaut -1, `waitpid()` attend la mort de n'importe quel fils (comme `wait()`).
• Le paramètre `status` a exactement le même rôle que `wait()`.
• Le paramètre `options` peut avoir les valeurs suivantes:
– **WNOHANG**: ne pas rester bloqué si aucun processus correspondant aux spécifications fournies par l'argument `pid` n'est terminé. Dans ce cas, `waitpid()` renverra 0.
– **WUNTRACED**: accéder également aux informations concernant les processus fils temporairement stoppés. C'est dans ce cas que les macros **WIFSTOPPED**(`statut`) et **WSTOPSIG**(`statut`) prennent leur signification.

La fonction `main()` d'un programme peut prendre des arguments en ligne de commande.

Exemple:

$ gcc mon_prog.c -o mon_prog

`On peut invoquer mon_prog avec des arguments:`

$ ./ mon_prog argument_1 argument_2 argument_3

Pour récupérer les arguments dans le programme **C**, on utilise les paramètres **argc** et **argv** de la fonction `main()`.

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

• L'entier **argc** donne le nombre d'arguments rentrés dans la ligne de commande $+1$.
• ***argv[]** est un pointeur sur un tableau de chaînes de caractères qui contient comme éléments:
– Le premier élément `argv[0]` est une chaîne qui contient le nom de fichier exécutable du programme.
– Les éléments `argv[1]`, `argv[2]`, `...` sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Exercice 3: Développer un programme en C qui prend des arguments et qui affiche le nom de fichier exécutable et les valeurs de ces arguments.

Les variables d'environnement sont définies sous la forme de chaînes de caractères contenant des affectations du type:

```
NOM=VALEUR
```

Ces variables sont accessibles aux processus. Lors de la duplication d'un processus avec un `fork()`, le fils hérite d'une copie des variables d'environnement de son père.

Un processus peut modifier, créer ou détruire des variables de son propre environnement. Un certain nombre de variables sont automatiquement initialisées par le système lors de la connexion de l'utilisateur.

Lorsqu'un programme C démarre, son environnement est automatiquement copié dans un tableau de chaînes de caractères. Ce tableau est disponible dans la variable globale **environ**, à déclarer ainsi en début de programme (elle n'est pas déclarée dans les fichiers d'entête courants):

```
char **environ;
```

Ce tableau contient des chaînes de caractères terminées par un caractère nul, et se finit lui-même par un pointeur nul.

Exemple:

```c
#include <stdio.h>
extern char **environ;
int main (void)
{
        int i = 0;
        for (i = 0; environ[i] != NULL; i ++)
        printf("%d : %s\n", i, environ[i]);
        return 0;
}
```

Variables d'environnement classiques couramment utilisées:

- **HOME**: contient le répertoire personnel de l'utilisateur.
- **PATH**: lorsqu'on tape la commande `ls`, par exemple, le système la cherche dans les répertoires spécifiés par la variable **PATH**, dans l'ordre où ils sont indiqués.
- **PWD**: contient le répertoire de travail.
- **USER** ou **LOGNAME**: nom de l'utilisateur.
- **TERM**: type de terminal utilisé.
- **SHELL**: shell de connexion utilisé.

Exemple:

```
$ echo $SHELL
```

Un processus au sein d'un système peut être indépendant ou coopératif. Un processus indépendant n'est pas affecté par l'exécution d'aucun processus. Tandis que, un processus coopératif peut être affecté par l'exécution d'autres processus.

La communication inter-processus (inter process communication "IPC") est un mécanisme qui permet aux processus de communiquer entre eux et de synchroniser leurs actions; en autre terme, c'est une méthode de coopération.

Avantages de l'IPC:

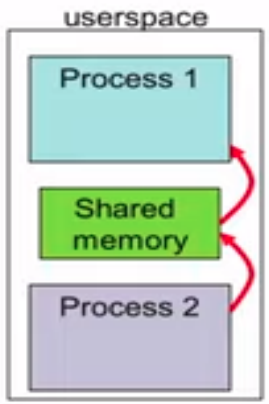
- Partage des informations.
- Partage des ressources.
- Augmentation la vitesse de calcul.
- Synchronisation entre les processus.
- ...

Les processus peuvent communiquer entre eux en utilisant:

- `la mémoire partagée (shared memory).`
- `la transmission de messages (message passing).`
- `Les signaux.`

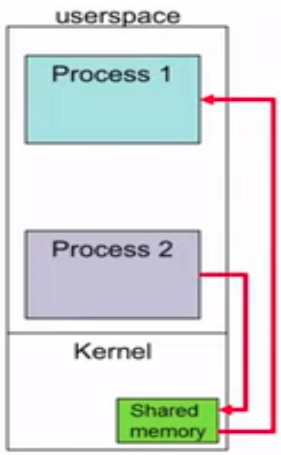
La mémoire partagée:

- Un processus va créer une zone en RAM à laquelle l'autre processus pourra accéder.
- Les deux processus peuvent accéder à la mémoire partagée comme une mémoire de travail normale.
  - lecture / écriture est comme lecture / écriture régulière.
  - rapide.
- Limitation: sujette aux erreurs, nécessite une synchronisation entre les processus.



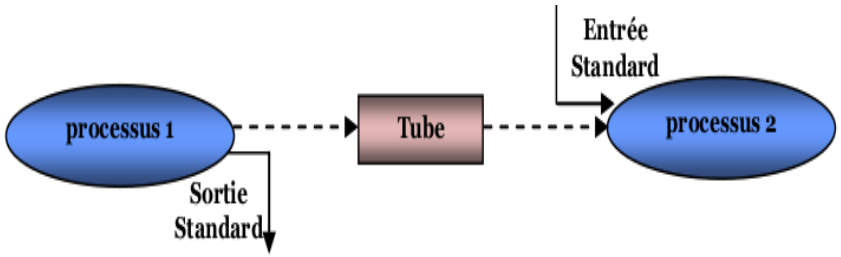
La transmission de messages:

- Mémoire partagée créée dans le noyau.
- Les appels systèmes tels que **send** et **receive** sont utilisés pour la communication.
  - `coopération`: chaque envoi doit avoir une réception.
- Avantage: moins sujette aux erreurs.
- Limitation: lente.



Un tube (en anglais pipe), espace géré par le noyau, peut être représenté comme un tuyau dans lequel circulent des informations. C'est une application de transmission de message.

Le principe des tubes est: la sortie standard d'un processus est redirigée vers l'entrée d'un tube dont la sortie est dirigée vers l'entrée standard d'un autre processus. Ainsi les deux processus peuvent échanger des informations sans passer par un fichier intermédiaire de l'arborescence. Cette communication est mono-directionnelle.

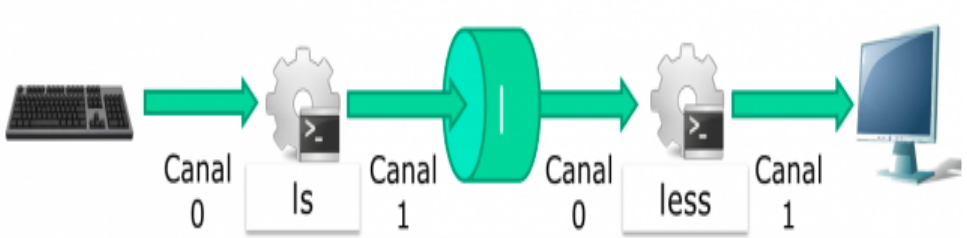


Le système Linux propose deux types de tubes: les tubes `anonymes` et les tubes `nommés`.

La création d'un tube anonyme se fait grâce à l'opérateur «|». Il est créé directement par le shell pour une commande donnée.

Syntaxe et exemple:

```
$ commandeA | commandeB
$ ls | wc
```

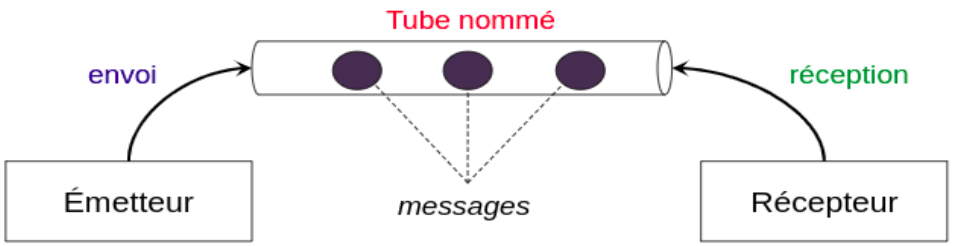


La commande `mkfifo` permet de créer un tube nommé.

Syntaxe et exemple:

```
mkfifo nomPipe
```

Un tube est un fichier spécial dans le système de fichiers.



- L'émetteur écrit dans le tube.
- Le récepteur lit à partir du tube.

Exemple:

```
# Crée un tube nommé
$ mkfifo pipe

# Écrit dans le tube nommé (terminal 1)
$ echo bonjour > pipe

# Lit à partir du tube nommé (terminal 2)
$ cat < pipe
```


EN PRÉPARATION


EN PRÉPARATION


EN PRÉPARATION

Mini projet 1: Compilation du noyau sous Linux

L'objectif de ce mini projet est de:

- savoir les avantages de la compilation du noyau (kernel) sous Linux.
- télécharger la dernière version du kernel Linux.
- configurer et compiler le kernel.
- préparer un rapport (en binôme) sur le mini projet.

Mini projet 2: Sécurité sous Linux

L'objectif de ce mini projet est de:

- ▷ présenter les concepts fondamentaux de la sécurité sous Linux.
- ▷ connaître les appels systèmes pour la sécurité sous Linux.
- ▷ implémenter la sécurité sous Linux.
- ▷ préparer un rapport (en binôme) sur le mini projet.