Système d'Exploitation (Operating System)

Linux –Distribution **Ubuntu**

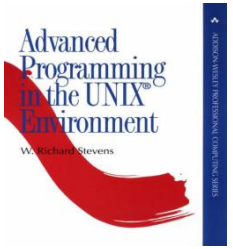
TP pour Licence 2 –Informatique, A.U. 2021–2022 –M. Taffar

Références Bibliographiques et Livres sur les Systèmes d'Exploitations

- Livres de Bases

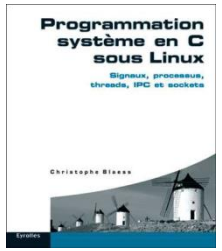
  - **Livre 1 :**

    Advanced programming in the UNIX environment, par Richard Stevens, Addison-Wesley, ISBN 0-201-56317-7



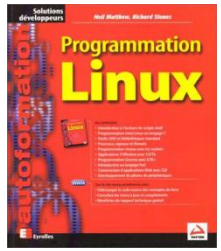
  - **Livre 2 :**

    Programmation Système en C sous Linux, par Christophe Blaess, Eyrolles, ISBN 2-212-11054-5



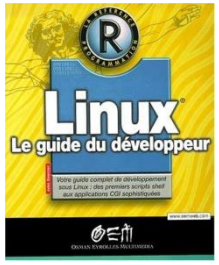
  - **Livre 3 :**

    Programmation Linux, par Neil Matthew et Richard Stones, Eyrolles, ISBN 1-861002-97-1



  - **Livre 4 :**

    Linux : Le guide du développeur, par John Goerzen, Eyrolles, ISBN 2-7464-0217-3



Références Webographiques Connexes aux Systèmes d'Exploitations

- Livres de Bases
  - **Livre 1** : https://www.fichier-pdf.fr/2014/04/03/linux-initiation-et-utilisation/linux-initiation-et-utilisation.pdf
  - **Livre 2** : https://repo.zenk-security.com/Linux%20et%20systemes%20.exploitations/Debuter%20sous%20Linux%20Edition%202.pdf
  - **Livre 3** : https://repo.zenk-security.com/Programmation/Programmation%20systeme%20en%20C%20sous%20Linux.pdf
  - **Livre 4** : https://tony3d3.free.fr/files/Programmation-Systeme-en-C-sous-Linux.pdf
  - **Livre 5** : ftp://hackbbs.org/Upload/Conception%20de%20syst%C3%A8mes%20d%E2%80%99exploitation%20-%20Le%20cas%20Linux.pdf
- N'hésitez surtout pas à consulter et interroger souvent Mr. ***Google*** !

Partie 1
Initiation Rapide & Prise en Main de Linux –Ubuntu

Mr M. Taffar
Dpt Informatique. Université de Jijel

TP pour Licence 2 –Informatique, A.U. 2021-2022

1. Introduction aux Systèmes d'Exploitations (SE)

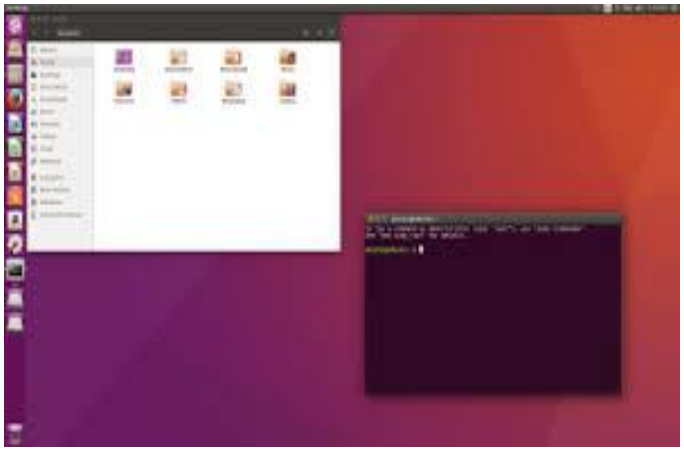
- Différents SE existent : Unix, Linux, OS, Windows, iOs, Android, Windows Phone, …
- OS (Mac, Apple) et Windows (Microsoft) : SE commerciaux (Copyright)
- Unix et Linux : SE open source (Copyleft) soumis à la licence GNU
- Windows et OS admettent plusieurs versions
- Linux : admet plusieurs distributions (RedHat, **Ubunto**, Debian, Fedora, Mandriva, …)
- Tous les laboratoires de recherche (en informatique), académies scientifique et universités du monde entier utilisent Linux.

2. Installation du Système Linux

- Différents environnements existent : Laptop, PC, station de travail, serveur, …
- Chaque environnement a sa propre configuration machine : il faut en tenir compte !
- Pour Linux la commande : *linuxconfig* permet d'avoir des infos sur la configuration
- Plusieurs Modes d'installation :
  - Linux seul sur une machine,
  - **Sous Machine virtuelle** (**voir étapes sur le site**),
  - Au choix au boot (au démarrage) avec un autre SE (ex. Windows ou Linux)

3. Commandes Linux de Base et Notion de Shell

- Linux dispose de 2 modes :
  - Mode textuel (ligne de commande avec prompt ou invite)
  - Mode graphique



- Il faut toujours afficher les 2 modes à côté l'un de l'autre (la fenêtre du répertoire de travail et la fenêtre de la ligne de commande)
- Le système Unix/Linux fait la différence entre *majuscule* et *minuscule* (ex. Paradigme ≠ PAradigme).

3. Commandes Linux de Base et Notion de Shell

- Après installation de Linux, toujours créer son compte utilisateur (répertoire *home*) puis son propre répertoire de développement.
- Ne jamais travailler au ***root*** (administrateur) –répertoire racine.
- Il faut toujours se connecter au compte utilisateur (user/home/~$) créer lors de l'installation (et ouvert en session user).
- Le compte *root* (racine: /$) ne doit servir qu'à l'administration du système.
- Il possible d'ouvrir une session sur sa propre machine ou sur une machine distante en utilisant telnet @IP de la machine distante
- **exit** : commande pour fermer la session courante
- **passwd** : pour modifier le mot de passe (au moins 6 caractères)

3.1. **man** (manuel)

- Commande ***man*** affiche l'aide en consultant les pages de manuel
- Pour obtenir de l'aide sur une commande, tapez :

  man <nom_commande>
- Exemple: ***man ls*** (pour quitter : Shift+q)
- L'aide sur une option : *man* ***a*** *ls* (donne les infos sur l'option ***a*** de ***ls***)
- L'option *--help* permet d'afficher l'aide sur une commande, en tapant :

  <nom_comande> --help

  exemple : ***ls*** **--help**

3.2. Types de commandes

- Trois types de commandes existent :
  - Pour **répertoires**,
  - Pour **fichiers**,
  - Pour **processus**
- Commandes pour Répertoires (directories)

cd	pwd	ls	mkdir	rmdir

- Commandes pour Fichiers (Files)

rm	cp	mv	find	chmod	ln	cat

- Commandes pour Processus, ex. : **ps** -user

3.3. Commandes pour Répertoires
ls / mkdir / rmdir / cd / pwd

Commande *ls*

- ***ls*** : permet de lister le contenu d'un répertoire (liste d'un répertoire)

Beaucoup d'options sont disponibles pour cette commande :

- **ls** (**sans option**) : liste les fichiers en plusieurs colonnes
- ***ls –l*** : liste les fichiers et répertoires avec les droits d'accès
- **ls –a** : liste tous les fichiers (même ceux commençant par un point)
- **ls | more** : liste écran par écran
- **NB**: La touche flèche "haut" du clavier vous permet de retrouver les commandes que vous avez déjà tapé.

Commandes : mkdir et rmdir

- **mkdir** : (make directory) création d'un répertoire

  mkdir <nom_répertoire> : permet de créer un répertoire

- **rmdir** : (remove directory) destruction d'un répertoire

  rmdir <nom_répertoire> : permet de supprimer un répertoire

Commande : cd (change directory)

- La commande **cd** permet de se déplacer dans les répertoires tant que les permissions l'autorisent
- Tapez **cd**
- **cd** <sans paramètre> : vous ramène dans votre répertoire personnel (rép. par défaut de l'utilisateur : **~** ou **$HOME**)
- **cd /home** : entrer dans le répertoire **/home** s'il existe !
- Tapez **cd ..** (puis **pwd**), que constateriez-vous ? (pour remonter au rép. parent)
- Tapez **cd /tmp** (puis pwd), que constateriez-vous ?
- Tapez **cd /** que constateriez-vous ? (pour remonter au rép. racine)
- Exemple : Créer un répertoire avec : **mkdir SE1G0**

  puis **cd SE1G0**

  puis **cd ..** ou **cd /**

  puis **cd** (rép. par défaut user **~** ou **$home**) puis **rmdir SE1G0**

Commande : pwd

- **pwd** : permet de savoir dans quel répertoire vous êtes ! (affiche le rép. courant)
- Copiez les fichiers d'un répertoire **/tmp/exos** dans votre répertoire courant avec la commande ***cp*** :

cp **/tmp/exos/*** **.** (le point final "." indique le répertoire courant)

- Utilisez : ***ls*** et ***ls –al*** (notez les différences d'affichages)
- Tapez **pwd**. Notez le répertoire dans lequel vous êtes.
- Tapez **cd /tmp** puis **pwd**. Notez le répertoire dans lequel vous êtes

3.4. Commandes pour Fichiers

cp / mv / find / chmod / ln / cat / more / less / diff / echo / rm

- Ces commandes ne concernent que les fichiers.
- Un fichier est un texte ASCII (code ou script) stocké sur un emplacement physique (sur le disque) et admet
  - *un lien physique (adresse ou adresse physique)* et
  - *un lien logique (son nom ou nom logique)*

Commandes pour Fichiers : cp / mv / find

- **cp** : permet de copier un ou plusieurs *fichiers* d'un emplacement (répertoire) vers un autre répertoire (dont il faut préciser le chemin d'accès : "***path***")

cp <nom_fichier> /home/perso

copie le fichier du répertoire courant vers le répertoire /home/perso

- **mv** : permet de déplacer un ou plusieurs *fichiers* d'un répertoire vers un autre

mv <nom_fichier> /home/perso

- **find** : permet de recherche ou trouver des fichiers depuis une racine spécifiée suivant des critères (**-name** : nom; **-mode** : dernière modification; …)
- **Exemple** : chercher des fichiers dans le répertoire courant qui contiennent la lettre ***f*** dans leurs noms, taper :

find . -name '*f*' -print

- **Exemple** : chercher des fichiers à partir de la racine root qui ont été modifiés il y a moins de 2 jours, taper :

find / -mtime -2

Commande pour Fichiers : chmod

- **chmod** : change les droits d'accès d'un fichier ou d'un répertoire *si vous en êtes le propriétaire ou le root*
- La syntaxe de chmod est :

chmod [a, u, g, o] [+ , -] [r , w , x] [nom_de_fichier]

- **L'argument qui donne les droits :**
  - a = (all) à tous les utilisateurs
  - u = (user) au propriétaire
  - g = (group) aux utilisateurs du groupe
  - o = (other) aux autres groupes

- **La valeur :**
  - \+ : ajouter (donner)
  - \- : enlevé (supprimer)
  - = : affectation

- **Les permissions autorisées :**
  - r = en lecture
  - w = en écriture
  - x = en exécution

- Avec la commande : ***ls -l*** les droits d'accès apparaissent en liste de 10 symboles :

  d rwx r-x r-x c'est-à-dire sous la forme : **s1 s2s3s4 s5s6s7 s8s9s10**

  – s1 : **-** (fichier classique) ou **d** (répertoire) ou **l** (lien symbolique)

- Exemples :
  – **chmod o-w fichier3** (enlève le droit d'écriture pour les autres)

Commandes pour Fichiers : ln / cat

- La commande ***ln*** : ajout de liens sur les fichiers
- ***ln*** permet d'ajouter un lien physique ou symbolique sur un fichier.

  ln <fichier.origine> <Nom.lien> : Crée un lien physique

  ln -s <fichier.origine> <Nom.lien> : Crée un lien symbolique
- Cette fonction est intéressante, elle permet d'avoir un seul fichier physique sur le disque et de le désigner sous plusieurs noms logiques.
- Cela peut, dans certains cas, faciliter les opérations de mises à jour ou de maintenance d'applications.
- **cat** : affichage ininterrompu d'un fichier
- **cat** permet d'afficher sans interruption un fichier ou plusieurs fichiers:

  cat fichier1 fichier2 (affiche fichier1 et fichier2)

Commandes pour Fichiers : diff / echo

- ***diff*** : affiche les différences entre deux fichiers
- Cela est parfois utilisé par des utilitaires stockant des modifications incrémentales dans des sources ou textes sous leur contrôle, permettant
  - de retrouver des versions antérieures ou
  - le travail à plusieurs
- ***echo*** : écrit sur la ligne courante les arguments qui lui sont passés

echo arg1 arg2 …

Exemple, tapez : **echo "Bonjour !"**

Commande pour Fichiers : rm

- **rm** : permet de détruire (supprimer) un ou plusieurs *fichiers* (liens logiques) ou *liens physiques* du répertoire courant (dans certains cas il faut préciser le chemin "*path*")

NB : Un fichier supprimé sous Linux ne peut pas être récupéré !

- **rm** <nom_fichier> : efface un fichier
- **rm** * : efface tous les fichiers du répertoire
- **rm** *–i* <fichier1> <fichier2> <fichier3> : l'option *–i* demande confirmation de l'effacement de chaque fichier

Commande : alias

- **alias** : donne toutes les commandes qui ont des alias
- Tapez : **alias echo**
- Pour ajouter un alias uniquement pour la session du terminal en cours : **alias elsada='echo'**
- Tapez : **alias elsada=echo**
- Tapez : **elsada="echo"**
- Que fait la commande : **elsada "ceci est un message écho !"** ?
- Pour retirer un alias, de la session du terminal en cours, saisir :

unalias elsada

- échapper un alias : lancer une commande sans que l'alias intervient

\echo

4. Le Multi-Processing (Multi-tâches)

- Multi-(processing/traitement/Utilisateur/Users/tâches/tasks/ processus/threads …)
- Unix/Linux est un SE multitâche multi-utilisateurs
- Un utilisateur peut **lancer plusieurs** tâches (programmes, processus ou commandes) **en parallèle** et même **en concurrence**
- Pour comprendre le concept de multitâche sous Linux, lancer la commande :

ps *-eaf*

4. Multi-processing

- On devrait obtenir des infos sous la forme :

UID	PID	PPID	C	STIME	TITY	TIME	CMD
root	1	0	0	08:27	?	00:00:04	init[5]

- Maintenant, lancer la commande : ***gedit*** puis **ps** ***-eaf*** (sur 2 fenêtres différentes)
- Chaque tâche (processus ou Cmd) appartient à quelqu'un (utilisateur) identifié par son **UID**
- Chaque processus est identifié par son numéro, c'est son **PID**
- Chaque processus est le fils d'un autre processus qui est identifié (le processus père) par son **PPID**
- Le reste des infos sont des ***infos de contrôle*** qui servent au scheduler (ordonnanceur de tâches)
- Pour n'avoir que les processus qui concernent l'utilisateur actuel, lancer la commande suivante ***: ps -eaf | grep login_utilisateur***

4.1. Commandes sur les Processus

- Lister vos processus à l'aide de la commande : ***ps -user***
  - Notez le **PID** de celui qui contient la chaine ***–bash***

- Tapez la commande: **kill -9 N°PID**
  - où **N°PID** est le numéro du processus que vous avez relevé.
  - Que s'est-il passé ?

- Tapez
  - la commande **clear** pour nettoyer l'écran puis
  - la commande **top**

Commande sur les Processus : *ps*

- ***ps*** : fournit la liste des processus actifs
- Tapez : ***ps***

  Cette commande écrite seule permet de lister les processus courant
- **ps -user**

  Liste les processus appartenant à l'utilisateur **user**
- **ps -aux**

  Liste de tous les processus du **système**
- **jobs -l**

  Liste des jobs avec leur numéro de job ainsi que leur numéro de processus système
- **NB** : la commande **top** : permet de connaître les processus gourmands en puissance de calcul.
- Ce qui nous intéresse c'est la première colonne PID, qui est le numéro des processus. C'est ce numéro qui est utilisé par **kill**

Commande sur les Processus : *kill*

- **kill** : cette commande permet de détruire (tuer, supprimer) un processus
- Exemple : si le processus cible est le numéro (PID) **546**

- **kill 546**

  Tente de détruire le processus 546

- **kill -9 546**

  Force la destruction du processus 546

5. Éditeur de texte sous Linux

- Comment écrire son programme ?
- Comme l'écriture de programmes et de scripts passe par l'utilisation d'un éditeur de texte, on se familiarisera avec l'éditeur ***gedit***
- D'autres éditeurs de textes sous Linux existent :
  - l'éditeur ***vi*** (éditeur archaïque en mode alphanumérique)
  - XEmacs
  - Emacs
  - *vim*

Principaux éditeurs de textes pour programmer sous Ubuntu

XEmacs

Emacs

vi

- Il en existe d'autres éditeurs : nedit, kwrite, kate, ...
- Choisissez celui qui vous convient le mieux !

6. Le Shell sous Linux

- Nous verrons comment utiliser **Shell** pour la programmation de *scripts*
- Un ***script*** est fichier contenant une série d'ordres ou commandes que l'on soumet à un ***programme*** externe pour qu'il les exécute.
- Ce ***programme*** est appelé ***interpréteur de commandes***
- ***Shell*** : est un interpréteur de commandes
- Un script doit être interprété (compilé + exécuté, commande par commande) dynamiquement
- **Shell** fait partie intégrante d'**Unix** depuis les débuts de celui-ci en 1971.

6.1. Le Shell Bash

- **Shell** est chargé de lire les *ordres que l'utilisateur saisit au clavier*, de les *analyser*, et *d'exécuter les commandes* correspondantes.
- Les tubes de communication (pipes) permettent de faire circuler des données de processus en processus
- le fichier *Shell* exécutable correspondant était traditionnellement **/bin/sh**
- Pour ***Shell C*** : Le fichier exécutable était ***/bin/csh***
- Avec la percée des logiciels libres, et du projet *GNU* (*Gnu is Not Unix*) destiné à cloner le système Unix, plusieurs distributions GNU/Linux ont vu le jour.
- La **FSF** (*Free Software Fundation*), fondée en 1985 par R. Stallman pour développer le projet GNU, donna naissance au fameux **shell *Bash***, que l'on trouve d'emblée sur les distributions Linux.
- **Bash** (acronyme de *Bourne Again Shell*) *: e*st un outil très puissant, que la plupart des utilisateurs de Linux emploient comme ***shell de connexion***.

6.2. La Norme Posix

- À la fin de 1980, il existait une pléthore de ***systèmes compatibles Unix*** qui implémentaient chacun de nouvelles fonctionnalités par rapport au système *Unix original*.
- Il était donc nécessaire d'uniformiser le paysage proposé par toutes ces versions d'Unix.
- Une norme fut proposée par l'IEEE : c'est la norme **Posix**.
- ***Posix*** : décrit l'interface entre le noyau du SE Unix et les applications, ainsi que le comportement d'un certain nombre d'outils système dont le ***shell***

6.3. *Pourquoi écrire un script shell ?*

- Pour lancer des commandes au prompt
- Parfois, entant qu'administrateur, pouvoir intervenir dans les scripts d'initialisation –de boot– du système pour personnaliser, par exemple, un serveur.
- Les scripts *shell* sont employés pour gérer des configurations de certaines applications nécessitant un paramétrage complexe,
- De nombreux serveurs utilisent une interface constituée de scripts shell pour transmettre des requêtes SQL au logiciel de base de données (*Oracle*, *MySQL*, etc.)

6.4. Exécution d'un script

- C'est de la **programmation par scripts**
- L'exécution d'un script Shell met en place des interactions entre le **shell**, le **noyau** et l'**interpréteur**
- Un script est exécuté par un **interpréteur**

6.5. Invocation de l'interpréteur

- Créons d'abord un scripts simple :
- Utilisez l'éditeur de texte *gedit* pour créer un fichier contenant la ligne suivante :

echo "Ceci est mon premier essai"

- La commande ***echo*** du shell affiche la chaîne transmise sur sa ligne de commande.
- À présent, exécuter le script **essai.sh** sur la ligne de commande en la précédant de l'**appel du shell** :

```
$ sh   essai.sh
Ceci est mon premier essai
$
```

- Le symbole $ correspond au symbole d'invite (prompt) du shell

6.5. Invocation de l'interpréteur

- **NB**:
  - le suffixe ***.sh*** du nom du script n'a pas d'importance ; c'est juste une info pour l'utilisateur.
  - Linux ne tient aucun compte des suffixes éventuels des noms de fichiers.
- Exemple, renommons-le et réessayons,

```
$ mv essai.sh essai
$ sh essai
Ceci est mon premier essai
$
```

- Testez :

```
$ sh test
/usr/bin/test: /usr/bin/test: cannot execute binary file
$
```

- **NB**: Cela est dû à la présence sur le système d'un autre fichier exécutable nommé test, mais qui n'est pas un script shell

6.6. Appel Direct

- Au lieu d'invoquer l'interpréteur suivi du nom du fichier, on préfère appeler directement le script, comme s'il s'agissait d'une commande habituelle
- Essayons :

```
$ essai
ksh: essai: non trouvé [Aucun fichier ou répertoire de ce type]
```

NB: le shell déplore de ne pouvoir trouver le fichier essai !

- Lorsqu'on invoque une commande Unix/Linux (ex., ***ls***), le système devra trouver un fichier exécutable de ce nom (exemple dans **/bin/ls**).
- Mais, impossible de parcourir tous les répertoires de tous les disques pour rechercher ce fichier, le temps de réponse serait très long !
- Linux ne recherche les fichiers implémentant les commandes dans un nombre restreint de répertoires.
- Ces répertoires sont énumérés dans la variable d'environnement **PATH** (en majuscules), que nous pouvons consulter le contenu par :

6.6. Appel Direct

- Examinons le contenu de la variable PATH, on la précédant par le symbole **$** :

```
$ echo   $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/home/cpb/bin
$
```

NB :

- Plusieurs répertoires sont indiqués, séparés par des deux-points.
- Le répertoire dans lequel nous nous trouvons ‘‘**$**’’ n’est pas mentionné, donc le système ne vient pas y chercher le fichier **essai**.

- Plusieurs possibilités peuvent être utilisées pour que le **shell** trouve le fichier **essai**

  - déplacer le script dans un des répertoires du PATH.
  - Si le script créé est utile pour tous les utilisateurs d’Unix/Linux, il faut le placer dans **/usr/bin**
  - Si ce script s’adressait uniquement aux utilisateurs de ce système, le répertoire **/usr/local/bin** serait plus approprié.
  - Si le script est réservé à votre usage personnel, le sous-répertoire **bin/** situé dans le répertoire personnel convient mieux.
  - Toutefois, s’il est en cours de développement, il vaut mieux le conserver dans un répertoire de travail spécifique et ajouter son chemin d’accès à la variable PATH

6.6. Appel Direct

- Modifier **PATH** en lui ajoutant un point (**.**) qui représente le répertoire courant

- Pour des raisons de sécurité, il faut **toujours** placer ce point en **dernière position** dans la variable PATH

- Exécuter la commande (**avant d'appeler essai**) :

```
$ PATH=$PATH:.
```

NB :

- On peut placer la commande dans le fichier d'initialisation (fichier ***profile***) du shell
- **mais**, ceci risque un problème de sécurité, si on travaille sur un système ouvert au public

6.6. Appel Direct

- Une autre (3ème) possibilité consiste à indiquer au système le répertoire où se trouve le fichier à chaque fois qu'on l'invoque.
- Ceci s'obtient facilement si le fichier se trouve dans le répertoire courant en utilisant l'appel :

```
$  ./essai
```

NB : Malheureusement, cela ne fonctionne pas !

```
$  ./essai
ksh: ./essai: ne peut pas exécuter [Permission non accordée]
$
```

- Le script ne s'exécute plus, **mais le message d'erreur a changé**
- Ceci est dû aux permissions accordées pour ce fichier, et que l'on peut observer avec l'option ***-l*** de ***ls*** :

```
$ ls  -l  essai
-rw-r--r-- 1 cpb users 34 sep 3 14:08 essai
$
```

6.6. Appel Direct

- Déchiffrant ce message :

-rw-r--r-- 1 cpb users 34 sep 3 14:08 essai

- La première colonne décrit le type de fichier et les permissions concernant sa manipulation :
  - Le 1er tiret **–** correspond à un fichier normal (**d** : répertoire, **l** : un lien symbolique, etc.).
  - Les 3 lettres suivantes indiquent que le propriétaire **cpb** du fichier a les droits de lecture (**r**) et d'écriture (**w**) sur son fichier, mais l'absence de lettre **x** (remplacée ici par un tiret ''**–**'') signifie que le propriétaire ne peut pas exécuter ce fichier. ***C'est ici le problème*** !
- Les trois lettres **r--** suivantes décrivent les droits donnés aux autres utilisateurs appartenant au même groupe que le fichier (users). Seule la lecture du contenu du fichier sera possible. Il ne sera pas possible de l'exécuter et de le modifier.
- Les 3 dernières lettres **r--** indiquent les permissions accordées à tous les autres utilisateurs du système (**r--** lecture seulement).

6.6. Appel Direct

- Le fichier n'étant pas exécutable, il est impossible de le lancer !
- Pour modifier les permissions d'un fichier, il faut utiliser **chmod**
- Il y a plusieurs d'options possibles pour **chmod**, prenant le cas suivant :

```
$ chmod +x essai
$ ls -l essai
-rwxr-xr-x 1 cpb users 34 sep 3 14:08 essai
$
```

- L'option **+x** de **chmod** ajoute l'autorisation d'exécution pour **tous les types d'utilisateurs**.
- Vérifiez que le script fonctionne :

```
$ ./essai
Ceci est mon premier essai
$ :
```

Test 1

- ✓ **Exo. 0 :** Créer dans le bureau un répertoire appelé **DONZ**, puis accéder à ce répertoire.
- ✓ **Exo. 1** : écrire un programme C qui calcule l'expression : **z =x*y+3**
- ✓ **Exo. 2** : Écrire un pgm qui vérifie si un variable **x** est **paire** ou **impaire**.
- ✓ **Exo. 3** : écrire un pgm qui calcul l'expression : $\mathbf{y = 2.x^2 + 3}$
- ✓ **Exo. 4** : écrie une fonction dans un pgm C qui compare 2 variables entières **a** et **b**.

Partie 2 : Programmation en C sous Linux –Ubuntu

Mr M. Taffar
Dpt Informatique. Université de Jijel

TP pour Licence 2 –Informatique, A.U. 2021-2022

1. Compilation et Modularité

- Cette partie concerne le développement en Langage C sous Unix/Linux –Ubuntu.
- Nous aborderons :
  - Comment compiler un programme C ?
  - Quelques options bien utiles
  - Directive de compilation
  - Découpage modulaire : exemple de module ‘‘mon_pgm’’

Compiler un programme C

- Le Compilateur **GCC** = **G**NU (OpenSource Project) **C** **C**ompiler
- C'est le **Compilateur C/C++ pour Linux**
- **gcc** s'exécute à partir de l'**interface en ligne de commande**
- Exemple de compilation :

```
/* mon_pgm.c */
int main(int argc, char ** argv) {
  printf ("Je suis las !\n");
  return 0;
}
```

```
[sysexploi@local]$ gcc mon_pgm.c
[sysexploi@local]$ ./a.out
Je suis las !
```

Quelques Options Utiles

- -o filename : permet de changer le nom du fichier de sortie (output).

```
[sysexploi@local]$ gcc mon_pgm.c  -o  monobjet
[sysexploi@local]$ ./monobjet
```

- -Wall : le compilateur effectue plus de vérification à la compilation et génère des *warnings* (avertissements) en cas de problème.

```
[sysexploi@local]$ gcc mon_pgm.c  -Wall  -o  monobjet

mon_pgm.c: In function ‘main’:
mon_pgm.c:4: warning: implicit declaration of function ‘printf’

[sysexploi@local]$ ./monobjet
Je suis las !
```

Directive de Compilation (1/2)

- ✓ **#include <filename>** : directive de compilation qui permet d'inclure le contenu d'un fichier.
- ✓ **gcc** va chercher le fichier **filename** dans les **répertoires de la librairie standard**.

```
/* mon_pgm.c */
#include <stdio.h>   /* stdio.h contient la déclaration de printf */

int main(int argc, char  **argv) {
   printf("Je suis las !\n");
   return 0;
}

[sysexploi@local]$ gcc mon_pgm.c  -Wall  -o  monobjet
[sysexploi@local]$ ./monobjet
Je suis las !
```

Directive de Compilation (2/2)

- ✓ **#include ``filename''** : directive de compilation qui permet d'inclure le contenu d'un fichier.
- ✓ **gcc** va chercher le fichier **filename** dans **le répertoire courant** (sinon préciser le chemin d'accès, home/se1/filename.h)

```
/* mon_pgm.h */
const char *hw = ''Je suis las !'';

/* mon_pgm.c */
#include <stdio.h>   /* stdio.h contient la déclaration de printf */
#include ''mon_pgm.h''

int main (int argc,  char  **argv) {
   printf("%s\n", hw);
   return 0;
}

[sysexploi@local]$ gcc  mon_pgm.c   -Wall   -o   monobjet
[sysexploi@local]$ ./monobjet
Je suis las !
```

Découpe Modulaire

- Un **module** : est un ensemble de *fonctions*, *types*, *variables* qui se concentre sur une partie du programme (ex., gestion d'un stack, d'un arbre, d'une BD, d'un périphérique USB, ...).
- Un **module** se compose généralement de **deux fichiers** :
  - un fichier **.h** et
  - un fichier **.c**
- Le fichier **.h** contient l'***interface du module*** (ce qui sera visible de l'extérieur du module) :
  - Les prototypes des fonctions
  - Les types, variables et constantes globales
- Le fichier **.c** contient l'***implémentation du module*** (cette implémentation n'est pas visible de l'extérieur) :
  - L'implémentation des fonctions
  - Les types, variables et constantes locales au module

Exemple de Module : *le Module mon_pgm*

```
---------------------------------------------------------- /* mon_pgm.h */
const  char  *hw = ‘‘Je suis las !’’;
void  print_je_suis_las();
---------------------------------------------------------- /* mon_pgm.c */
#include <stdio.h>   /* stdio.h fichier librairie du compilateur C contient la déclaration de printf */
#include ‘‘mon_pgm.h’’ /* mon_pgm.h fichier librairie/bibliothèque utilisateur contient la
                                déclaration de la variable hw et le prototype de print_je_suis_las() */
void print_je_suis_las() {
    printf("%s\n", hw);
}
---------------------------------------------------------- /* pgmprinc.c  ou main.c */
#include "mon_pgm.h"
int main (int argc,  char  **argv) {
     print_je_suis_las();
     return 0;
}
-------------------------------------------------------------------------------------------------------------------------------
[sysexploi@local]$ gcc  mon_pgm.c   -Wall   -o   monobjet1
[sysexploi@local]$ gcc  pgmprinc.c   -Wall   -o   monobjet2
[sysexploi@local]$ gcc  mon_pgm.c   pgmprinc.c   -Wall   -o   monobjet3
                                          /*pgmprinc.c  avec  et sans #include mon_pgm.h */
[sysexploi@local]$ ./monobjet3
Je suis las !
```

Exemple de Module : *Options de Compilation*

[sysexploi@local]$ **gcc mon_pgm.c -Wall -o mobj1** /* Référence à ***main()*** non définie */
[sysexploi@local]$ **gcc pgmprinc.c -Wall -o mobj2** /*Référence indéfinie à **print_je_suis_las()** */

1ère solution : utiliser l'option : —c

[sysexploi@local]$ **gcc mon_pgm.c -c -Wall "-o mobj1"** /*Compile le module et génère l'objet par défaut **mon_pgm.o** */

[sysexploi@local]$ **gcc -c -Wall pgmprinc.c "-o mobj2"** /*pgmprinc.c inclut mon_pgm.h, génère l'objet par défaut **pgmprinc.o** */

[sysexploi@local]$ gcc **mon_pgm.o pgmprinc.o** -o monobjet

[sysexploi@local]$ gcc **mobj2 mobj1** -o monobjet

[sysexploi@local]$ ./monobjet

2ème solution : utiliser l'option : —c

[sysexploi@local]$ **gcc -c pgmprinc.c mon_pgm.c -Wall -o mobj**
/***pgmprinc.c** avec ou sans **mon_pgm.h** */

[sysexploi@local]$ **gcc -c pgmprinc.c mon_pgm.c mon_pgm.h -Wall -o mobj**
/***pgmprinc.c** n'inclut pas **mon_pgm.h** */

[sysexploi@local]$ ./mobj

Je suis las !

2. Problème d'inclusion et Solution

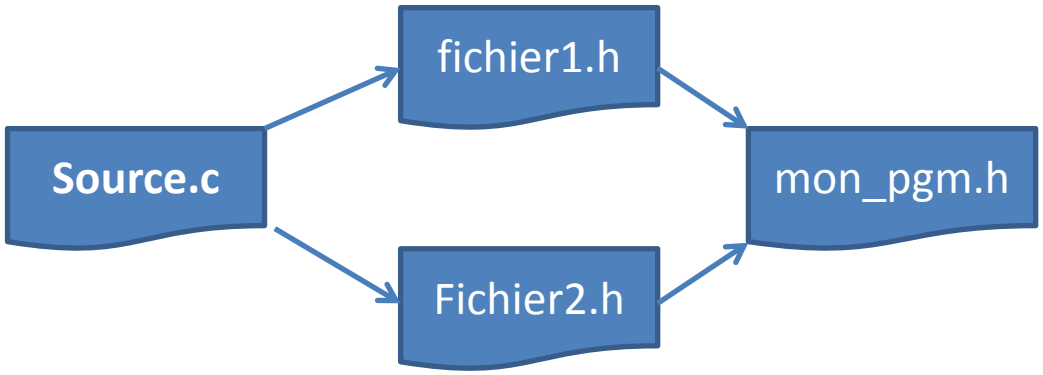
- Nous étudions :
  - Problème d'inclusion multiple
  - D'autres directives à la rescousse !
  - Protection contre l'inclusion multiple

Problème d'inclusion multiple

- Supposons que dans un fichier ***source.c*** se trouve inclus deux fichiers :

  fichier1.h et fichier2.h

- Et que par ailleurs, ces 2 fichiers aient besoin de faire appel à notre module *mon_pgm*, définit dans **mon_pgm.h**
- Ce fichier "**mon_pgm.h**" sera inclus deux fois dans **source.c** (une fois via *fichier1.h* et une fois via *fichier2.h*).
- Cela peut causer des problèmes à la compilation !!!



D’autres directives à la rescousse !

- Il faut donc protéger les fichiers **.h** contre les inclusions multiples.
- Celle-là est réalisée en utilisant les directives de compilation

  #define et #ifndef ... #endif

- #define permet de définir un symbole qui sera remplacé à la compilation

```
#define PI 3.141592654
```

- #ifndef ... #endif permet d’inclure du code selon qu’un symbole a été défini précédemment ou pas

```
#ifndef SYMBOLE
    /* Code à inclure si SYMBOLE n’est pas défini */
#endif
```

Protection contre l'inclusion multiple

- Lors de la première inclusion on définit un symbole unique pour le fichier **.h**
- Puis, à chaque inclusion, on teste l'existence du symbole.
- Le module *mon_pgm* revisité :

```
/* mon_pgm.h */
#ifndef __MON_PGM_H
#define __MON_PGM_H
const char *hw = "Je suis las !";
void print_je_suis_las();
#endif
```

3. Compilation Séparée & Fichier Makefile

✓ Cette séance comprend :

- Comment effectuer une compilation séparée ?
- Comment automatiser la compilation ?
- Un exemple de **Makefile**

Compilation séparée (1)

- Pour compiler un programme composé de plusieurs modules, on peut passer tous les fichiers **.c** en option à **gcc** :

```
[sysexploi@local]$ gcc -Wall mon_pgm.c main.c -o monobjet
[sysexploi@local]$ ./monobjet
Je suis las !
```

- En utilisant cette méthode, si on modifie un module, on doit tout recompiler !
- Heureusement, **gcc** offre une alternative : la compilation séparée avec l'option **–c**
- Chaque fichier **.c** est compilé en un fichier **.o**

```
[sysexploi@local]$ gcc -Wall -c mon_pgm.c
[sysexploi@local]$ gcc -Wall -c main.c
```

Compilation séparée (2)

- Pour créer l'exécutable, il ne reste plus qu'à faire l'édition des liens (*linkage*) :

```
[sysexploi@local]$ gcc mon_pgm.o main.o -o monobjet
[sysexploi@local]$ ./monobjet
Je suis las !
```

- Ainsi, lorsqu'un module est modifié, il suffit de recompiler le module modifié uniquement, et ensuite d'effectuer l'édition des liens

Automatiser la Compilation

- Le problème de la compilation séparée, c'est qu'il faut recompiler à la main chaque fichier modifié : ça peut vite devenir long et fastidieux !
- Il est possible d'automatiser le processus de compilation en utilisant l'outil **make**
- Pour cela, il faut décrire dans un fichier, appelé ***Makefile*** ou ***makefile***, les règles de compilation.
- Chaque règle est de la forme :

<*fichier voulu*> : <*fichiers dont il dépend*>

<*command pour créer le fichier*>

Exemple de Makefile

- Reprenons l'exemple du module *mon_pgm*

```
# Makefile pour le module mon_pgm
# (le caractère '#' dénote un commentaire)

monobjet:   main.o   mon_pgm.o
        gcc   main.o   mon_pgm.o   -o   monobjet

mon_pgm.o:  mon_pgm.c   mon_pgm.h
        gcc   -Wall  -c  mon_pgm.c

main.o:  main.c  mon_pgm.h
        gcc  -Wall  -c  main.c
```

- Lancer la compilation et l'édition de liens en tapant : **make**

4. Gestion d'Erreurs en C

- Séance dédiée principalement aux points liés à la ***gestion des erreurs dans un programme C*** sous Linux.

La Gestion d'Erreurs en C (1/3)

- Les *appels système* ainsi que les *fonctions* des librairies standards C sont susceptibles de *ne pas fonctionner comme attendu* :
  - ouverture (**fopen()** ou **open()**) d'un fichier alors que l'utilisateur n'en a pas les permissions,
  - allocation (**malloc**) de mémoire alors qu'il n'y a plus assez de mémoire disponible
  - Et tant d'autres, ...
- Le langage C offre toute une série de fonctions/variables, permettant une gestion d'erreur propre, telles que :
  - *errno*
  - *strerror*
  - *perror*
  - *assert*

La Gestion d'Erreurs en C (2/3)

- La variable entière **errno**

Elle est mise à jour par les appels système et par certaines fonctions des librairies standard C. En cas d'erreur, elle contient un code d'erreur détaillant ce qui s'est mal passé :

```
#include  <errno.h>
extern  int  errno;
```

- La fonction **strerror()**

Elle permet d'obtenir à partir d'un code d'erreur (venant de **errno**) un message plus compréhensible

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

La Gestion d'Erreurs en C (3/3)

- La fonction **perror()**

Elle permet d'afficher un message d'erreur sur le standard error (= msg) suivit du détail de la dernière erreur rencontrée durant un appel système ou un appel à une fonction des librairies standards :

```
#include <stdio.h>
void perror(const char *msg);
```

- La macro **assert()**

Elle termine l'exécution du programme si <*expression*> est évaluée à faux (0). Ces gardes peuvent être désactivée en définissant le symbole **NDEBUG** avant d'inclure ***assert.h*** (en utilisant **#define**)

```
#include <assert.h>
void assert(int expression);
```

5. Debugger

- Nous développons les points liés au débogage (exécuter, repérer et enlever les erreurs ou les dysfonctionnements) des programmes C :
  - Le tracing
  - Le *tracing* conditionnel
  - Débogueur symbolique
  - Le GDB : Debogueur symbolique pour linux

Le Tracing (tracer/traçage)

- Le tracing consiste à ajouter des ***printf*()** dans le code pour afficher des *messages de débogage* (valeur d'une variable, résultat de fonction, ...).
- Cela permet de savoir pendant l'exécution où l'on se trouve dans le code :

```
int main(int argc, char  **argv) {
    int x, y;
    /* du code jouant avec x et y */
    if (x < y) {
        printf(‘‘main: (x < y)\n’’);     fflush(stdout);
    }
    else {
        printf(‘‘main: (y >= x)\n’’);    fflush(stdout);
    }
    return 0;
}
```

Tracing Conditionnel

- Malheureusement, les ***printf***() peuvent vite devenir nombreux et gênant, surtout lorsqu'on a fini de déboguer le programme.
- On peut résoudre ce problème en utilisant les directives de compilation **#ifdef ... #endif** et **#define**

```
#ifdef __DEBUG
printf(‘‘Mon message de débogage !\n’’); fflush(stdout);
#endif
```

- Pour activer les messages de débogage, il suffit de définir la constante voulue (**__DEBUG** dans l'exemple) à l'aide de la directive **#define**
- On peut également utiliser un symbole de debug différent pour chaque module.

Débogueur symbolique

- Une autre façon de déboguer un programme est d'utiliser un débogueur symbolique.
- Ce type de débogueur permet :
  - d'exécuter un programme **pas à pas**
  - d'afficher les **valeurs intermédiaires** des variables
  - de placer des ***breakpoint*** (*points d'arrêts*)
  - d'examiner le **stack** (la pile) **des appels de fonction**
  - et d'autres ...
- D'une manière plus générale, il permet de comprendre et surtout de localiser l'erreur (*segmentation fault*) !

Débogueur GNU –GDB (1/3)

- GDB = **G**NU **D**e**B**ugger
- Débogueur symbolique pour linux
- Interface interactive, en mode texte
- Avant de l'utiliser, il est nécessaire de compiler le programme avec l'option **–g**, afin de rajouter des informations symboliques dans l'exécutable (noms des fonctions, variables, ...)

```
[sysexploi@local]$ gcc -Wall –g bug.c –o bug
```

GDB (2/3)

- Ensuite, on peut lancer GDB en passant l'exécutable en paramètre :

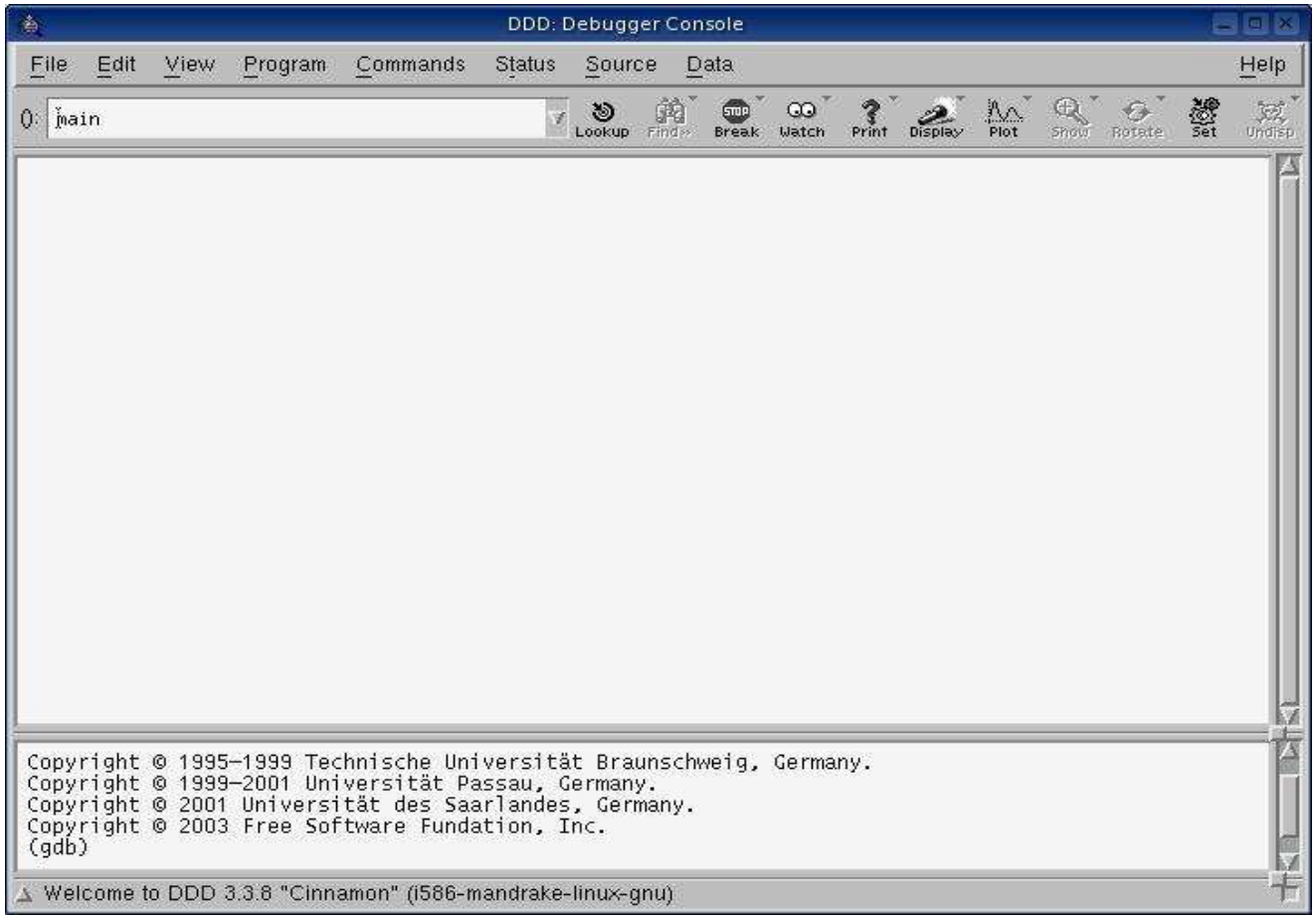
```
[cmeuter@litpc34 error]$ gdb bug
GNU gdb 6.0-2mdk (Mandrake Linux)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty“ for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb)
```

GDB (3/3)

- Les commandes les plus utilisées de GDB :
  - ✓ **l** : affiche la ligne de code courante, ainsi que les 5 lignes suivantes et précédentes.
  - ✓ **l num** : affiche la ligne de code **num**.
  - ✓ **b num** : place un ***break point*** à la ligne de code **num**.
  - ✓ **n** : exécute l'instruction suivante sans entrer dans les appels de fonctions.
  - ✓ **s** : exécute l'instruction suivante et entre dans les appels de fonctions.
  - ✓ **p expression** : affiche le résultat de l'évaluation de **expression**.
  - ✓ **r args** : execute le programme chargé avec les arguments **args**
  - ✓ **bt** : affiche le stack d'appels de fonctions.

Interface à GDB

- Il existe des interfaces graphiques à GDB.
- On peut citer DDD = Data Display Debugger



Test 2

1. Écrire un module (fichiers **.h** et **.c**) comprenant une fonction ou procédure qui calcule l'expression : $\mathbf{a^b}$, avec a et b des entiers positifs.
   a) Le prototype de la fonction sera déclaré dans "**puiss.h**".
   b) L'implémentation de la fonction dans "**puiss.c**".
   c) La fonction main dans le fichier "**princ.c**".
2. Écrire un programme composé d'un module et d'un pgm principal qui calcul l'expression : $\mathbf{(a^n+b)^m}$, avec a, b, n et m des entiers positifs.
3. Écrire un *makefile* qui permet de compiler le module et le programme en utilisant la compilation séparée.

Test 3

1. Écrire un module (fichiers **.h** et **.c**) de gestion d'une liste dynamique d'entiers, simplement chaînée, comprenant :
   a) Une fonction d'initialisation.
   b) Une fonction d'insertion en début de liste.
   c) Une fonction d'insertion en fin de liste.
   d) Une fonction pour effacer une liste.
2. Écrire un programme de *démo* qui utilise les fonctionnalités du module.
3. Écrire un *Makefile* qui permet de compiler le module et le programme démo en utilisant la compilation séparée.

Partie 3 : Programmation Système en C sous Linux –Ubuntu

Mr M. Taffar

Dpt Informatique. Université de Jijel

TP pour Licence 2 –Informatique, A.U. 2021-2022

Objectif du TP

- Le but est de présenter :
  - Les principes généraux de la *programmation sous Linux*, ainsi que
  - Les outils disponibles pour réaliser des *applications système*.
- Nous nous concentrerons sur
  - La programmation système et
  - La manipulation des processus (tels les appels : **fork**, **wait**, **kill**, **sleep**, etc.)
- Nous nous familiariserons également aux utilitaires et bibliothèques permettant la programmation système en C sous Linux

1. Génération d'un Processus (par l'appel : fork)

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

- La seule manière, ensuite, de créer un nouveau processus est d'appeler le système par l'appel-système **fork()**, qui va dupliquer le processus appelant.

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

1. Génération d'un Processus (par l'appel : fork)

- L'appel-système **fork()**
  - est déclaré dans **<unistd.h>**
  - par la fonction : **pid_t fork(void);**

- Pour connaître son propre identifiant **PID**, on utilise l'appel-système ***getpid*()**, qui ne prend pas d'argument et renvoie une valeur de type **pid_t**.

- Il s'agit, bien entendu, du PID du **processus appelant**.

1. Génération d'un Processus (par l'appel : fork)

- Cet appel-système, déclaré dans <unistd.h>, est l'un des rares qui n'échouent jamais : **pid_t *getpid*(void);**

- La fonction **fork** retourne :
  - **–1** en cas d'erreur ;
  - Sinon, **0** dans le **processus fils** et le **PID du fils** dans le **processus père**.

- Ceci permet au père de connaître le **PID** de **son fils**.

1. Génération d'un Processus (par l'appel : fork)

Exercice 1

- Ecrire un programme C qui crée **deux processus** (**un père** et **son fils**).
- Chacun d'eux affichera à l'écran qui il est :
  - Le **père** écrit : ‘‘je suis le père de PID : **?**’’,
  - Le **fils** écrit : ‘‘je suis le fils de PID : **?** et mon père est de PID : **?**’’).

1. Génération d'un Processus (par l'appel : fork)

Solution Exo. 1

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

int main(void)  {
                pid_t pid_fils;
                pid_fils = fork();

                if (pid_fils == -1) {
                                puts("Erreur de création du nouveau processus");
                                exit (1);
                }
                if (pid_fils == 0) {
                                printf("Nous sommes dans le fils\n"); /* la fonction getpid permet de connaître son propre PID*/
                                printf("PID du fils est \%d\n", getpid()); /* fonction getppid permet de connaître le PPID (PID de son père) */
                                printf("Le PID de mon père (PPID) est \%d", getppid());
                }
                else {
                                printf("Nous sommes dans le père\n");
                                printf("Le PID du fils est \%d\n", pid_fils);
                                printf("Le PID du père est \%d\n", getpid());
                                printf("PID du grand-père : \%d", getppid());
                }
                return 0;
}
```

1. Génération d'un Processus (par l'appel : fork)

Exercice 2

- Ecrire un programme C qui crée **deux processus fils**.
  - L'un affiche les entiers de **1 à 50**, et
  - L'autre affiche les entiers de **51 à 100**.

1. Génération d'un Processus (par l'appel : fork)

Solution Exo. 2

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

int main(void) {
        pid_t pid;
        int i;

        if ((pid = fork()) == -1) { perror("fork"); exit(1); }
        if (pid == 0) { /* fils1 */
                for (i = 1; i <= 50; i++) printf("%d\n", i);
                return 0;
        }

        if ((pid = fork()) == -1) { perror("fork"); exit(1); }
        if (pid == 0) { /* fils2 */
                for (i = 51; i <= 100; i++) printf("%d\n", i);
                return 0;
        }
        return 0;
}
```

2. Terminaison d'un Processus Fils

États d'un Processus

- **Actif** (**R**): en cours d'exécution (running)
- **Prêt/En Attente/Arrêt** (waiting, stopped) (**T**): prêt à être exécuté (runnable) ou stoppé/arrêté **temporairement** (et attend un SIGCONT)
- **Endormi/Bloqué** (en veille ou en sommeil, sleeping) (**S**): interrompu, en attente non-active/passive, mais peut être réveillé par un évènement
- **Sommeil ininterrompu** (attente active) (**D**): à l'arrêt (mais actif ! Ex. USB bloqué)
- **Zombie** (ou défunt, defunct/zombi) (**Z**) : **pour un processus fils**
- **Terminer** (finished/terminate) : Mort (et détruit, destroy)

2. Terminaison d'un Processus Fils (par l'appel : wait)

- Lorsque le processus **fils** se termine (soit en sortant du ***main*** soit par un appel à ***exit*()**) avant le processus père, le processus fils ne disparaît pas complètement, mais devient un ***zombie***

- Pour permettre à un processus fils à l'état de ***zombie*** de **disparaître complètement**, le processus père peut appeler l'instruction "**wait(NULL);**"

- L'appel **wait()** se trouve dans la bibliothèque "**sys/wait.h**"

2. Terminaison d'un Processus Fils (par l'appel : wait)

Attention

l'appel de **wait** est **bloquant**, c'est-à-dire que lorsque la fonction **wait** est appelée, l'exécution du **père est suspendue** jusqu'à ce qu'un fils se termine

Idée

Ainsi, **il faut mettre autant d'appels de *wait*() qu'il y a de fils**

2. Terminaison d'un Processus Fils (par l'appel : wait)

- La fonction **wait** renvoie le code d'erreur **–1** dans le cas où le **processus n'a pas de fils**.

- **wait()** est fréquemment utilisée pour permettre au processus père ***d'attendre la fin de ses fils*** avant de se terminer lui-même

Par exemple, pour récupérer le résultat produit par un fils

- Il est possible de mettre le processus père en attente de la fin d'un processus fils **particulier** par l'instruction :

waitpid(pid_fils, NULL, 0);

2. Terminaison d'un Processus Fils (par l'appel : wait)

Exercice 3

- Ecrire un programme C qui crée **un processus fils**, le père devra attendre la fin du fils à l'aide de la primitive ***wait*()** afin qu'il puisse récupère le **code renvoyé par le fils** dans la fonction ***exit*()**.

2. Terminaison d'un Processus Fils (par l'appel : wait)

Solution Exo. 3

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h> /* permet de récupérer les codes d'erreur */

int main(void) {
        pid_t pid_fils;
        int status;
        switch (pid_fils=fork())
        {
                case -1 :       perror("Problème dans fork()\n");
                                exit(errno); /* retour du code d'erreur ou encore utiliser exit(1) */
                                break;
                case 0 :        puts("Je suis le fils");
                                puts("Je retourne le code 3");
                                exit(3);  //ou encore exit(2);
                default :       printf("Je suis le père : j'ai lancé le processus fils %d\n'', pid_fils);
                                puts("Je récupère le code de retour\n");
                                wait(&status);
                                puts(''Mon fils s'est terminé normalement !!\n");
                                printf(''Le code de sortie du fils %d, son statut est : %d\n", pid_fils, WEXITSTATUS(status));
                                break;
        }
        return 0;
}
```

3. Endormir un processus avec sleep()

- La fonction la plus simple pour ***endormir*** ***temporairement*** un processus est **sleep()**

- **sleep()** est déclarée dans le fichier bibliothèque **<unistd.h>** par le prototype :

  unsigned int ***sleep*** *(unsigned int* ***nb_secondes****);*

- Cette fonction **endort le processus** pendant la durée demandée et **revient ensuite** (puis le réveille)

3. Endormir un processus (avec l'appel : sleep())

- **sleep()** : pour **endormir le processus** pour une durée déterminée.
- sleep() endort le **processus appelant**
  - jusqu'à ce que **nb_sec** secondes se soient écoulées, ou
  - jusqu'à ce qu'un **signal non ignoré soit reçu**.
- **sleep**() renvoie (valeur de retour) :
  - **zéro** si le temps prévu s'est écoulé, ou
  - le **nombre de secondes restantes** si l'appel a été interrompu par un gestionnaire de signal.

3. Endormir un processus (avec l'appel : sleep())

Exercice 4

- Ecrire un programme en C qui crée **deux processus** (un **père** et **son fils**).
- Le processus **père** dort **deux (02) secondes** avant d'envoyer un signal à son fils.
- Ce dernier (le fils) essaye de **dormir 10s**, mais sera réveillé plus tôt par le signal.
- On invoque la commande système «**date**» pour afficher l'heure avant et après l'appel **sleep**(). On présente également la durée restante.

3. Endormir un processus (avec l'appel : sleep())

Solution Exo. 4

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

void gestionnaire_sigusr1 ( int numero ) {
}

int main (void) {
        pid_t pid;
        unsigned int duree_sommeil;
        struct sigaction action;

        if ((pid = fork()) < 0) {
                fprintf(stderr, "Erreur dans fork \n");
                exit(EXIT_FAILURE);
        }
        action.sa_handler = gestionnaire_sigusr1;
        sigemptyset (&action.sa_mask);
        action.sa_flags = SA_RESTART;
```

Solution Exo. 4 (suite)

```
        if (sigaction(SIGUSR1, &action, NULL) != 0) {
                fprintf (stderr, "Erreur dans sigaction \n");
                exit(EXIT_FAILURE);
        }

        if (pid == 0) {
                system("date +\"%H:%M:%S\"");
                duree_sommeil = sleep(10);
                system ("date +\"%H:%M:%S\"");
                fprintf (stdout, "Durée restante %u\n", duree_sommeil);
        }
        else {
                sleep(2);
                kill(pid, SIGUSR1);
                waitpid(pid, NULL, 0);
        }

        return EXIT_SUCCESS;
}
```

Test 4

- **Exo. 1** : Écrire un pgm qui utilise une fonction permettant de permuter 2 variables **a** et **b**. Le pgm doit comporter 3 fichiers :
  - **Interface1.h**
  - **Interface1.c**
  - **Pgmprinc.c**

- **Exo. 2** : Transformer le pgm C précédent pour que : la variable **a** sera fournit par le processus **fils1** et la variable **b** par le processus **fils2**, tandis que le processus **père** fait la permutation.