

# Programming Shell



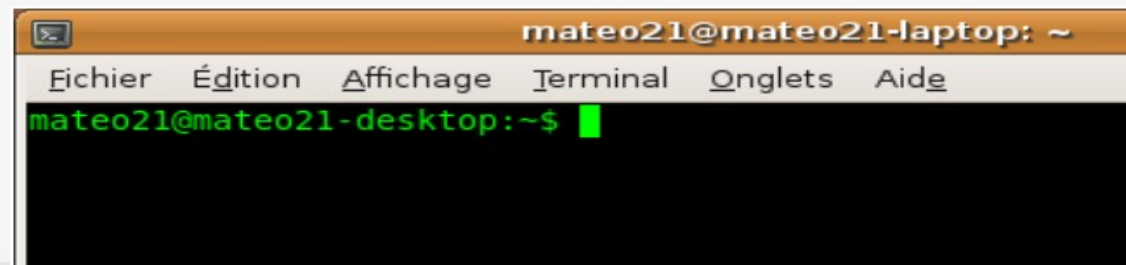
# C'est quoi le Shell ?

- ❖ Supposons **un mini-langage de programmation intégré à Linux** qui n'est pas un langage aussi complet comme par exemple le C, le C++ ou le Java, mais cela permet d'automatiser la plupart de vos tâches : **sauvegarde des données, surveillance de la charge de votre machine, etc.....**
- ❖ Vous pouvez faire tout cela en créant un programme en C par exemple. **Le gros avantage du langage shell est d'être totalement intégré à Linux : il n'y a rien à installer, rien à compiler.** Et surtout : vous avez très peu de nouvelles choses à apprendre. En effet, toutes les commandes que l'on utilise dans les scripts shell sont des commandes du système que vous connaissez déjà : ls, cut, grep, sort...



# C'est quoi le Shell ?

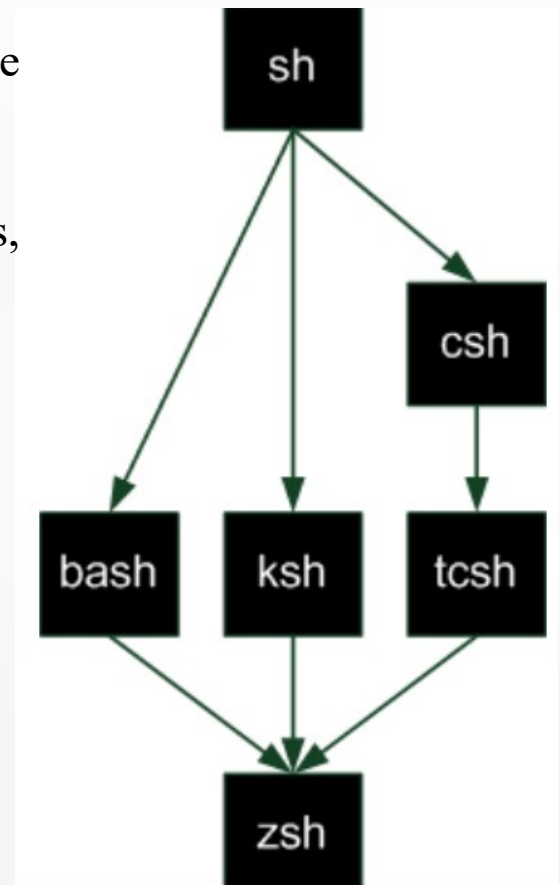
- ❖ L'interprète de commandes (**shell**) permet d'interagir avec le système; et exécute des commandes (**modification / consultation de l'état du système**)
- ❖ Il y'a deux environnements très différents disponibles sous Linux :
  - ✓ **L'environnement console**
  - ✓ **L'environnement graphique.**
- ❖ La plupart du temps, sur sa machine, on a tendance à utiliser l'environnement graphique, qui est plus intuitif. Mais, la console est aussi un allié très puissant qui permet d'effectuer des actions habituellement difficiles à réaliser dans un environnement graphique.



# Types de Shell

Voici les noms de quelques-uns des principaux **shells** qui existent :

- ✓ **sh** : Bourne Shell. L'ancêtre de tous les shells, et il est installé sur tous les OS basés sur Unix. Il est néanmoins pauvre en fonctionnalités par rapport aux autres shells.
- ✓ **bash** : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.
- ✓ **ksh** : Korn Shell. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- ✓ **csch** : C Shell. Un shell utilisant une syntaxe proche du langage C.
- ✓ **tcsh** : Tenex C Shell. Amélioration du C Shell.
- ✓ **zsh** : Z Shell. Shell assez récent reprenant les meilleures idées de **bash**, **ksh** et **tcsh**.



# Description de Shell

- ❖ Autocompléter une commande ou un nom de fichier lorsque vous appuyez sur Tab (figure suivante) ;
- ❖ Gérer les processus (envoi en arrière-plan, mise en pause avec Ctrl + Z...) ;
- ❖ Rediriger et chaîner les commandes (les fameux symboles >, <, |, etc.) ;
- ❖ Définir des alias (par exemple ll signifie chez moi ls -lArth).

⇒ Bref, le shell fournit toutes les fonctionnalités de base pour pouvoir lancer des commandes.



# Programmation de Shell

## ➤ Installation de Shell :

Vous devriez avoir **sh** et **bash** installés sur votre système. Si vous voulez essayer **un autre shell**, comme **ksh** par exemple, vous pouvez le télécharger comme n'importe quel paquet :

```
# apt-get install ksh
```

Une fois installé, il faut demander à l'utiliser pour votre compte utilisateur. Pour cela, tapez :

```
$ chsh
```

=> **chsh** signifie **Change Shell**.

On vous demandera où se trouve le programme qui gère **le shell**. Vous devrez indiquer /  
**bin/ksh pour ksh, /bin/sh pour sh, /bin/bash pour bash. On va se concentrer plus**  
**sur le bash.**



# Programmation de Shell

## ➤ Création de script Bash :

Nous allons commencer par étudier un premier script **bash** tout simple via lequel on pourra voir les bases de la création d'un script et comment celui-ci s'exécute, pour cela on crée un nouveau fichier pour notre script. Le plus simple est d'ouvrir **Vim** en lui donnant le nom du nouveau fichier à créer :

```
$ vim essai.sh
```

Si `essai.sh` n'existe pas, il sera créé (ce qui sera le cas ici). L'extension **.sh** indique le script **shell**, Certains scripts **shell** n'ont d'ailleurs pas d'extension du tout.

=> Le script `essai` doit contenir une syntaxe du programmation du shell **bash** qui est le plus répandu sous Linux et plus complet que **sh**. Nous indiquons où se trouve le programme **bash**

```
#!/bin/bash
```

La ligne du sha-bang **#!** permet donc de « charger » le bon shell avant l'exécution du script, vous devrez la mettre au tout début de chacun de vos scripts.



# Programmation de Shell

## ➤ Exécution de commandes :

Après le sha-bang `#!/`, vous pouvez commencer à coder et lancer les commandes dans le script **essai.sh**

```
#!/bin/bash
```

```
ls
```

Pour les commentaires, vous pouvez les ajouter dans le script. Ce sont des lignes qui ne seront pas exécutées mais qui permettent d'expliquer ce que fait votre script.

Tous les commentaires commencent par un `#`. Par exemple :

```
#!/bin/bash
```

```
# Affichage de la liste des fichiers
```

```
ls
```





# Programmation de Shell

## ➤ Exécution du script bash :

Pour enregistrer votre fichier et fermez votre éditeur. Sous **Vim**, il suffit de taper **:wq** ou encore **:x**. Le script s'exécute maintenant comme n'importe quel programme, en tapant « **./** » devant le nom du script :

```
$ ./essai.sh  
essai.sh
```

« Ce script fait juste un ls, il affiche donc la liste des fichiers présents dans le répertoire »

Pour exécuter un script, **il faut que le fichier ait le droit « exécutable »**. Le plus simple pour donner ce droit est d'écrire :



```
$ chmod +x essai.sh
```

# Programmation de Shell

## ➤ Les variables en bash :

### a) Déclaration d'une variable :

- ✓ La variable a pour **nom** **message** ;
- ✓ Et pour **valeur** « **Bonjour tout le monde** » :

```
$ ./essai.sh  
essai.sh
```

- Ne mettez pas d'espaces autour du symbole égal « = »
- Si vous voulez insérer une apostrophe dans la valeur de la variable, il faut la faire précéder avec un antislash \.



```
message='Bonjour c\'est moi'
```

# Programmation de Shell

## ➤ Les variables en bash :

Comme n'importe quel langage de programmation, on trouve en **bash** ce que l'on appelle **des variables**. Ces variables nous permettent **de stocker temporairement des informations en mémoire. Ce qui est la base de la programmation.**

### a) Déclaration d'une variable :

On peut créer un nouveau script que nous appellerons **variables.sh** par l'instruction suivante

```
$ vim variables.sh
```

La première ligne de tous nos scripts doit indiquer quel shell est utilisé, et commence par cette phrase

```
#!/bin/bash
```

=> ce qui indique qu'on va programmer avec le bash

On définit une variable. Toute variable possède un nom et une valeur :

```
message='Bonjour tout le monde'
```



# Programmation de Shell

## ➤ Les variables en bash :

### a) Déclaration d'une variable :

- ✓ Si on reprend notre script. Il devrait à présent ressembler à ceci :

```
#!/bin/bash  
  
message='Bonjour tout le monde'
```

- Si on l'exécute pour voir ce qui se passe (après avoir modifié les droits pour le rendre exécutable, bien sûr) => rien ne se passe :

```
$ ./variables.sh  
$
```



Ce script met en mémoire le message « **Bonjour tout le monde** », et c'est tout ! Rien ne s'affiche à l'écran ! Pour afficher une variable, il va falloir utiliser la commande

« **echo** »

# Programmation de Shell

## ➤ Les variables en bash :

### b) Affichage d'une variable :

- ✓ La commande « **echo** » affiche dans la console le message demandé. Un exemple :

```
$ echo Salut tout le monde  
Salut tout le monde
```

- la commande « **echo** » affiche dans la console tous les paramètres qu'elle reçoit. Ici, nous avons envoyé quatre paramètres : Salut ; tout ; le ; monde.
- Chacun des mots était considéré comme un paramètre affiché. Si vous mettez des guillemets autour de votre message, celui-ci sera considéré comme étant un seul et même paramètre (le résultat sera visuellement le même) :



```
$ echo "Salut tout le monde"  
Salut tout le monde
```

```
$ echo -e "Message\nAutre ligne"  
Message  
Autre ligne
```

# Programmation de Shell

## ➤ Les variables en bash :

### b) Affichage d'une variable :

- ✓ Pour afficher une variable, on peut de nouveau utiliser son nom précédé du symbole dollar \$ :

```
#!/bin/bash  
  
message='Bonjour tout le monde'  
echo $message
```

- Mais, supposons que l'on veuille afficher à la fois du texte et la variable. Nous serions tentés d'écrire :

```
#!/bin/bash  
  
message='Bonjour tout le monde'  
echo 'Le message est : $message'
```

le résultat est :

```
Le message est : $message
```

```
message='Bonjour tout le monde'  
echo "Le message est : $message"  
Le message est : Bonjour tout le monde
```



# Programmation de Shell

## ➤ Les variables en bash :

### c) Demande d'une saisie :

- ✓ Vous pouvez demander à l'utilisateur de saisir du texte avec la commande **read** :
- ✓ Adaptons ce script qui nous demande de saisir un nom puis qu'il nous l'affiche :

```
#!/bin/bash  
  
read nom  
echo "Bonjour $nom !"
```

- Lorsque vous lancez ce script, rien ne s'affiche, mais vous pouvez taper du texte (Mathieu, par exemple) :



```
Mathieu  
Bonjour Mathieu !
```

# Programmation de Shell

## ➤ Les variables en bash :

### c) Demande d'une saisie :

- **-p : afficher un message de prompt :** pour rendre l'utilisateur conscient de ce qu'il va saisir, par exemple :

```
#!/bin/bash  
  
read -p 'Entrez votre nom : ' nom  
echo "Bonjour $nom !"
```

- Lorsque vous lancez ce script, vous aurez ce résultat :



```
Entrez votre nom : Mathieu  
Bonjour Mathieu !
```



# Programmation de Shell

## ➤ Les variables en bash :

### c) Demande d'une saisie :

- **-n : limiter le nombre de caractères :** pour limiter à l'utilisateur le nombre de caractères qu'il va saisir, par exemple :

```
#!/bin/bash  
  
read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom  
echo "Bonjour $nom !"
```

- Lorsque vous lancez ce script, vous aurez ce résultat :



```
Entrez votre login (5 caractères max) : mathiBonjour mathi !
```

# Programmation de Shell

## ➤ Les variables en bash :

### c) Demande d'une saisie :

- **-n : limiter le nombre de caractères :** => Notez que le bash coupe automatiquement au bout de 5 caractères sans que vous ayez besoin d'appuyer sur la touche Entrée; mais on peut faire par exemple :

```
#!/bin/bash

read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo -e "\nBonjour $nom !"
```



- Lorsque vous lancez ce script, vous aurez ce résultat :

```
Entrez votre login (5 caractères max) : mathi
Bonjour mathi !
```

# Programmation de Shell

## ➤ Les variables en bash :

### c) Demande d'une saisie :

- **-s : ne pas afficher le texte saisi** : il permet de masquer les caractères que vous saisissez.

Cela vous servira notamment si vous souhaitez que l'utilisateur entre un mot de passe :

```
#!/bin/bash

read -p 'Entrez votre mot de passe : ' -s pass
echo -e "\nMerci ! Je vais dire à tout le monde que votre mot de passe est $pass ! :)"
```

- Lorsque vous lancez ce script, vous aurez ce résultat :



```
Entrez votre mot de passe :
Merci ! Je vais dire à tout le monde que votre mot de passe est supertopsecret38 ! :)
```

# Programmation de Shell

## ➤ Les variables en bash :

### d) Les opérations mathématiques :

- En bash, les variables sont toutes des chaînes de caractères. Il n'est pas vraiment capable de manipuler des nombres ; il n'est donc pas capable d'effectuer des opérations, il possible de passer par des commandes à titre d'exemple **let**

```
#!/bin/bash  
  
let "a = 5"  
let "b = 2"  
let "c = a + b"  
echo $c
```

- Lorsque vous lancez ce script, la variable \$c vaut :

7



# Programmation de Shell

## ➤ Les variables en bash :

### d) Les opérations mathématiques :

- Les opérations utilisables sont :

- l'addition : + ;
- la soustraction : - ;
- la multiplication : \* ;
- la division : / ;
- la puissance : \*\* ;
- le modulo (renvoie le reste de la division entière) : %

```
let "a = 5 * 3" # $a = 15
let "a = 4 ** 2" # $a = 16 (4 au carré)
let "a = 8 / 2" # $a = 4
let "a = 10 / 3" # $a = 3
let "a = 10 % 3" # $a = 1
```

-N.B :  $10 / 3 = 3$  car la division est entière (la commande ne renvoie pas de nombres décimaux) ;

- $10 \% 3$  renvoie 1 car le reste de la division de 10 par 3 est 1. En effet, 3 « rentre » 3 fois dans 10 (ça fait 9), et il reste 1 pour aller à 10.



# Programmation de Shell

## ➤ Les variables en bash :

### d) Les opérations mathématiques :

- Il est possible aussi de contracter les commandes, comme cela se fait en langage C.

```
let "a = a * 3"
```

- Equivalent à :

```
let "a *= 3"
```

**N.B : Les résultats renvoyés sont des nombres entiers et non des nombres décimaux. Si vous voulez travailler avec des nombres décimaux, renseignez-vous sur le fonctionnement de la commande `bc`**



# Programmation de Shell

## ➤ Les variables en bash :

### e) Les variables d'environnement :

```
$ env
ORBIT_SOCKETDIR=/tmp/orbit-mateo21
GLADE_PIXMAP_PATH=/usr/share/glade3/pixmaps
TERM=xterm
SHELL=/bin/bash
GTK_MODULES=canberra-gtk-module
USER=mateo21
PATH=/home/mateo21/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
GDM_XSERVER_LOCATION=local
PWD=/home/mateo21/bin
EDITOR=nano
SHLVL=1
HOME=/home/mateo21
OLDPWD=/home/mateo21
```

[ ... ]

- Actuellement, les variables que vous créez dans vos scripts bash n'existent que dans ces scripts.

=> C-à-d une variable définie dans un programme A ne sera pas utilisable dans un programme B.

- Les variables d'environnement sont des variables que l'on peut utiliser dans n'importe quel programme. On parle aussi parfois de variables globales. Vous pouvez afficher toutes celles que vous avez actuellement en mémoire avec la commande env

# Programmation de Shell

## ➤ Les variables en bash :

### e) Les variables d'environnement :

- Il y en a beaucoup. Certaines sont très utiles, d'autres moins. Parmi celles on trouve :

**SHELL** : indique quel type de shell est en cours d'utilisation (sh, bash, ksh...) ;

**PATH** : une liste des répertoires qui contiennent des exécutables que vous souhaitez pouvoir lancer sans indiquer leur répertoire. Nous en avons parlé un peu plus tôt. Si un programme se trouve dans un de ces dossiers, vous pourrez l'invoquer quel que soit le dossier dans lequel vous vous trouvez ;

**EDITOR** : l'éditeur de texte par défaut qui s'ouvre lorsque cela est nécessaire ;

**HOME** : la position de votre dossierhome ;

**PWD** : le dossier dans lequel vous vous trouvez ;

**OLDPWD** : le dossier dans lequel vous vous trouviez auparavant.



=> Notez que les noms de ces variables sont, par convention, écrits en majuscules.



# Programmation de Shell

## ➤ Les variables en bash :

### e) Les variables d'environnement :

- Il y en a beaucoup. Certaines sont très utiles, d'autres moins. Parmi celles on trouve :
- ⇒ Notez que les noms de ces variables sont, par convention, écrits en majuscules.
- par exemple ;

```
#!/bin/bash
```

```
echo "Votre éditeur par défaut est $EDITOR"
```

Le résultat est :

```
Votre éditeur par défaut est nano
```

```
$ env
ORBIT_SOCKETDIR=/tmp/orbit-mateo21
GLADE_PIXMAP_PATH=/usr/share/glade3/pixmaps
TERM=xterm
SHELL=/bin/bash
GTK_MODULES=canberra-gtk-module
USER=mateo21
PATH=/home/mateo21/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
GDM_XSERVER_LOCATION=local
PWD=/home/mateo21/bin
EDITOR=nano
SHLVL=1
HOME=/home/mateo21
OLDPWD=/home/mateo21
[ ... ]
```

# Programmation de Shell

## ➤ Les variables en bash :

### e) Les variables des paramètres :

- Vos scripts bash peuvent eux aussi accepter des paramètres. On pourrait appeler le script comme ceci :

```
./variables.sh param1 param2 param3
```

- Le problème, c'est que nous n'avons toujours pas vu comment récupérer ces paramètres dans notre script. Mais il y a la méthode suivante :

En effet, des variables sont automatiquement créées :

- ✓ \$# : contient le nombre de paramètres ;
- ✓ \$0 : contient le nom du script exécuté (ici ./variables.sh) ;
- ✓ \$1 : contient le premier paramètre ;
- ✓ \$2 : contient le second paramètre ;
- ✓ ... ;
- ✓ \$9 : contient le 9e paramètre.

```
#!/bin/bash
```

```
echo "Vous avez lancé $0, il y a $# paramètres"  
echo "Le paramètre 1 est $1"
```



```
$ ./variables.sh param1 param2 param3  
Vous avez lancé ./variables.sh, il y a 3 paramètres  
Le paramètre 1 est param1
```

# Programmation de Shell

## ➤ Les variables en bash :

### e) Les tableaux :

- Le bash gère également les variables « tableaux ». Ce sont des variables qui contiennent plusieurs cases, comme un tableau. Pour définir un tableau, on crée une variable tableau qui contient trois valeurs ( valeur0, valeur1, valeur2 ).

```
tableau=('valeur0' 'valeur1' 'valeur2')
```

- Pour accéder à une case du tableau, il faut utiliser la syntaxe suivante :

```
echo ${tableau[2]}
```

```
tableau[2]='valeur2'
```

```
echo ${tableau[*]}
```

```
valeur0 valeur1 valeur2 valeur5
```



# Programmation de Shell

## ➤ Les conditions en bash :

- La syntaxe de condition en bash est la suivante fi est la fin de condition :

```
if [ test ]
```

```
then
```

ou

```
if [ test ]; then
```

```
echo "C'est vrai"
```

```
fi
```

```
echo "C'est vrai"
```

```
fi
```

- Voici un exemple sur un script que nous appellerons conditions.sh :

Comme **\$nom** égale à « Bruno », ce script affichera :

```
#!/bin/bash
```

```
nom="Bruno"
```

```
if [ $nom = "Bruno" ]
```

```
then
```

```
echo "Salut Bruno !"
```

```
fi
```

```
Salut Bruno !
```



# Programmation de Shell

## ➤ Les conditions en bash :

- Vous pouvez tester deux variables à la fois dans le **if** :

```
#!/bin/bash

nom1="Bruno"
nom2="Marcel"

if [ $nom1 = $nom2 ]
then
    echo "Salut les jumeaux !"
fi
```

```
#!/bin/bash

nom="Bruno"

if [ $nom = "Bruno" ]
then
    echo "Salut Bruno !"
else
    echo "J'te connais pas, ouste !"
fi
```

- Comme **\$nom1** est différent de **\$nom2**, le contenu du **if** ne sera pas exécuté. Le script n'affichera donc rien.

# Programmation de Shell

## ➤ Les conditions en bash :

- On propose aussi plutôt de se baser sur le premier paramètre (\$1) envoyé au script :

```
#!/bin/bash

if [ $1 = "Bruno" ]
then
    echo "Salut Bruno !"
elif [ $1 = "Michel" ]
then
    echo "Bien le bonjour Michel"
elif [ $1 = "Jean" ]
then
    echo "Hé Jean, ça va ?"
else
    echo "J'te connais pas, ouste !"
fi
```

- On teste le script par insertion d'une variable paramètre et on obtient ce résultat

```
$ ./conditions.sh Bruno
Salut Bruno !
```

Mais, si on met autre chose on obtient

```
$ ./conditions.sh Jean
J'te connais pas, ouste !
```

# Programmation de Shell

## ➤ Les tests de if en bash :

### a) Tests sur des chaînes de caractères :

- En bash toutes les variables sont considérées comme des chaînes de caractères. Il est donc très facile de tester ce que vaut une chaîne de caractères par exemple :

```
#!/bin/bash

if [ $1 != $2 ]
then
    echo "Les 2 paramètres sont différents !"
else
    echo "Les 2 paramètres sont identiques !"
fi
```

```
$ ./conditions.sh Bruno Bernard
Les 2 paramètres sont différents !
```

```
$ ./conditions.sh Bruno Bruno
Les 2 paramètres sont identiques !
```

Condition	Signification
<code>\$chaine1 = \$chaine2</code>	Vérifie si les deux chaînes sont identiques. Notez que bash est sensible à la casse : « b » est donc différent de « B ». Il est aussi possible d'écrire « == » pour les habitués du langage C.
<code>\$chaine1 != \$chaine2</code>	Vérifie si les deux chaînes sont différentes.
<code>-z \$chaine</code>	Vérifie si la chaîne est vide.
<code>-n \$chaine</code>	Vérifie si la chaîne est non vide.

# Programmation de Shell

## ➤ Les tests de if en bash :

### a) Tests sur des chaînes de caractères :

- Par exemple :

```
#!/bin/bash

if [ -z $1 ]
then
    echo "Pas de paramètre"
else
    echo "Paramètre présent"
fi
```

```
$ ./conditions.sh
Pas de paramètre
```

```
$ ./conditions.sh param
Paramètre présent
```

Condition	Signification
<code>\$chaine1 = \$chaine2</code>	Vérifie si les deux chaînes sont identiques. Notez que bash est sensible à la casse : « b » est donc différent de « B ». Il est aussi possible d'écrire « == » pour les habitués du langage C.
<code>\$chaine1 != \$chaine2</code>	Vérifie si les deux chaînes sont différentes.
<code>-z \$chaine</code>	Vérifie si la chaîne est vide.
<code>-n \$chaine</code>	Vérifie si la chaîne est non vide.



# Programmation de Shell

## ➤ Les tests de if en bash :

### b) Tests sur des nombres :

- Bien que bash gère les variables comme des chaînes de caractères pour son fonctionnement interne, mais ça nous vous empêche pas de faire des comparaisons de nombres si ces variables en contiennent.

Condition	Signification
<code>\$num1 -eq \$num2</code>	Vérifie si les nombres sont égaux ( <b>eq</b> ual ). À ne pas confondre avec le « = » qui, lui, compare deux chaînes de caractères.
<code>\$num1 -ne \$num2</code>	Vérifie si les nombres sont différents ( <b>n</b> on <b>eq</b> ual ). Encore une fois, ne confondez pas avec « != » qui est censé être utilisé sur des chaînes de caractères.
<code>\$num1 -lt \$num2</code>	Vérifie si num1 est inférieur ( < ) à num2 ( <b>l</b> ower <b>t</b> han ).
<code>\$num1 -le \$num2</code>	Vérifie si num1 est inférieur ou égal ( <= ) à num2 ( <b>l</b> ower or <b>eq</b> ual ).
<code>\$num1 -gt \$num2</code>	Vérifie si num1 est supérieur ( > ) à num2 ( <b>g</b> reater <b>t</b> han ).
<code>\$num1 -ge \$num2</code>	Vérifie si num1 est supérieur ou égal ( >= ) à num2 ( <b>g</b> reater or <b>eq</b> ual ).

# Programmation de Shell

## ➤ Les tests de if en bash :

### b) Tests sur des nombres :

- Par exemple

```
#!/bin/bash

if [ $1 -ge 20 ]
then
    echo "Vous avez envoyé 20 ou plus"
else
    echo "Vous avez envoyé moins de 20"
fi
```

```
$ ./conditions.sh 23
Vous avez envoyé 20 ou plus
```

```
$ ./conditions.sh 11
Vous avez envoyé moins de 20
```



# Programmation de Shell

## ➤ Les tests de if en bash :

### c) Tests sur les fichiers :

- Un des avantages de bash par rapport à d'autres langages est que l'on peut très facilement faire des tests sur des fichiers

Condition	Signification
-e \$nomfichier	Vérifie si le fichier existe.
-d \$nomfichier	Vérifie si le fichier est un répertoire. N'oubliez pas que sous Linux, tout est considéré comme un fichier, même un répertoire !
-f \$nomfichier	Vérifie si le fichier est un... fichier. Un vrai fichier cette fois, pas un dossier.
-L \$nomfichier	Vérifie si le fichier est un lien symbolique (raccourci).
-r \$nomfichier	Vérifie si le fichier est lisible (r).
-w \$nomfichier	Vérifie si le fichier est modifiable (w).
-x \$nomfichier	Vérifie si le fichier est exécutable (x).

# Programmation de Shell

## ➤ Les tests de if en bash :

### c) Tests sur les fichiers :

- Par exemple

```
#!/bin/bash

read -p 'Entrez un répertoire : ' repertoire

if [ -d $repertoire ]
then
    echo "Bien, vous avez compris ce que j'ai dit !"
else
    echo "Vous n'avez rien compris..."
fi
```

```
Entrez un répertoire : /home
Bien, vous avez compris ce que j'ai dit !
```

```
Entrez un répertoire : rienavoir.txt
Vous n'avez rien compris...
```



# Programmation de Shell

## ➤ Les tests de if en bash :

### d) Tests multiples :

- Dans if, il est possible de faire plusieurs tests à la fois. En général, on vérifie :

- si un test est vrai ET qu'un autre test est vrai; par le symbole : **&&**
- si un test est vrai OU qu'un autre test est vrai; par le symbole : **||**
- Inversion du test par **!**

```
if [ ! -e fichier ]
then
    echo "Le fichier n'existe pas"
fi
```

```
#!/bin/bash
if [ $# -ge 1 ] && [ $1 = 'koala' ]
then
    echo "Bravo !"
    echo "Vous connaissez le mot de passe"
else
    echo "Vous n'avez pas le bon mot de passe"
fi
```

- Le test vérifie deux choses :

- qu'il y a au moins un paramètre ( « si \$# est supérieur ou égal à 1 » ) ;
- que le premier paramètre est bien koala ( « si \$1 est égal à koala » ).

Le résultat et le suivant :

```
$ ./conditions.sh koala
Bravo !
Vous connaissez le mot de passe
```



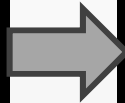
# Programmation de Shell

## ➤ Le case « plusieurs conditions à la fois » :

- On reprend l'exemple précédent et on le refait par le case conditionnel Si la condition est vérifiée, tout ce qui suit est exécuté jusqu'au prochain double point-virgule ;;

```
#!/bin/bash

if [ $1 = "Bruno" ]
then
    echo "Salut Bruno !"
elif [ $1 = "Michel" ]
then
    echo "Bien le bonjour Michel"
elif [ $1 = "Jean" ]
then
    echo "Hé Jean, ça va ?"
else
    echo "J'te connais pas, ouste !"
fi
```



```
#!/bin/bash

case $1 in
    "Bruno")
        echo "Salut Bruno !"
        ;;
    "Michel")
        echo "Bien le bonjour Michel"
        ;;
    "Jean")
        echo "Hé Jean, ça va ?"
        ;;
    *)
        echo "J'te connais pas, ouste !"
        ;;
esac
```

# Programmation de Shell

## ➤ Le case « plusieurs conditions à la fois » :

- On peut faire des « ou » dans un case. Dans ce cas, petit piège, il ne faut pas mettre deux || mais un seul ! Exemple :

```
#!/bin/bash

case $1 in
    "Chien" | "Chat" | "Souris")
        echo "C'est un mammifère"
        ;;
    "Moineau" | "Pigeon")
        echo "C'est un oiseau"
        ;;
    *)
        echo "Je ne sais pas ce que c'est"
        ;;
esac
```



# Programmation de Shell

## ➤ Les boucles :

### b) La boucle while :

Le type de boucle que l'on rencontre le plus couramment en bash est la boucle **while**.

```
while [ test ]  
do  
    echo 'Action en boucle'  
done
```

ou

```
while [ test ]; do  
    echo 'Action en boucle'  
done
```

Par exemple on veut demander à l'utilisateur de dire « oui » et répéter cette action tant qu'il n'a pas fait ce que l'on voulait. Nous allons créer un script **boucles.sh** pour cet exemple : On fait deux tests. Est-ce que \$reponse est vide \$reponse est différent de oui ? tant que l'un des deux tests est vrai, on recommence la boucle.

```
#!/bin/bash  
  
while [ -z $reponse ] || [ $reponse != 'oui' ]  
do  
    read -p 'Dites oui : ' reponse  
done
```

Le résultat  
sera le suivant

```
Dites oui : euh  
Dites oui : non  
Dites oui : bon  
Dites oui : oui
```

Il existe aussi le mot clé **until**, qui est l'inverse de **while**. Il signifie « **Jusqu'à ce que** ». Il peut être utilisé aussi dans le code shell



# Programmation de Shell

## ➤ Les boucles :

### b) La boucle for :

- La boucle **for** permet de parcourir une liste de valeurs et de boucler autant de fois qu'il y a de valeurs.

```
#!/bin/bash

for variable in 'valeur1' 'valeur2' 'valeur3'
do
    echo "La variable vaut $variable"
done
```

Ce qui donne  
ce résultat

```
La variable vaut valeur1
La variable vaut valeur2
La variable vaut valeur3
```

- La variable va prendre successivement les valeurs **valeur1,valeur2,valeur3**. La boucle va donc être exécutée trois fois et la variable vaudra à chaque fois une nouvelle valeur de la liste.

```
#!/bin/bash

liste_fichiers=`ls`

for fichier in $liste_fichiers
do
    echo "Fichier trouvé : $fichier"
done
```

Ce qui donne  
ce résultat

```
Fichier trouvé : boucles.sh
Fichier trouvé : conditions.sh
Fichier trouvé : variables.sh
```


# Programmation de Shell

## ➤ Les fonctions :

- Une fonction, c'est un ensemble d'instructions, permettant d'effectuer plusieurs tâches avec des paramètres d'entrée différents. Son utilisation vous permettra de rendre votre programme plus lisible et structuré. Ainsi, il facilitera le développement de votre programme.

### ❖ *Comment déclarer une fonction ?*

- En Bash, il y a deux manières pour déclarer une fonction :



```
# déclaration méthode 1
maFonction ()

{
  bloc d'instructions
}

#appel de ma fonction
maFonction
```

Ou

```
# déclaration méthode 2

function maFonction

{
  bloc d'instructions
}

#appel de la fonction
maFonction
```

# Démarrage du système Linux

## ❖ Processus du démarrage :

### I. Bios

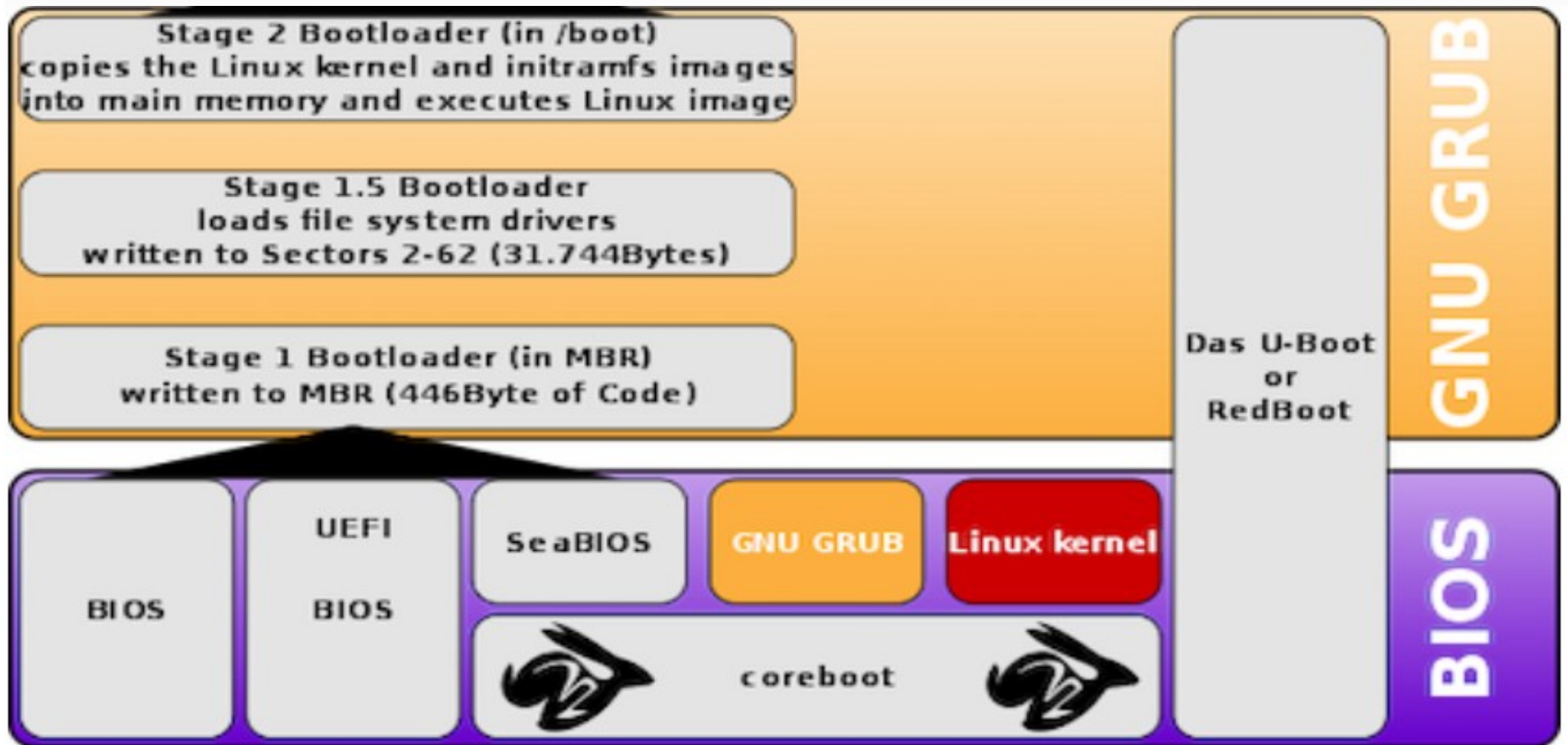
- **Le BIOS - Basic Input Output System (système d'entrée sortie de base)** est essentiel à tout PC, il se trouve généralement dans une mémoire morte ou ROM qui est directement implantée sur la carte mère du PC. Il est associé à une mémoire sauvegardée par une petite pile bouton sur la carte mère (le “setup” qui est la sauvegarde de la configuration).
- Si votre PC ne démarre pas (ou ne boot pas) c'est à cause du BIOS et de sa configuration (le setup). On peut accéder au setup et le modifier en pressant une touche dès la mise sous tension du PC.
- Le BIOS teste le matériel et y applique les réglages mémorisés dans le setup, tout en s'assurant qu'il n'existe pas de dysfonctionnement matériel et que tout est présent dans la machine, mémoire CPU principalement.
- il regarde la présence des périphériques nécessaires au boot : lecteur de disquette, cd rom, disque dur.... si vous avez installé un système d'exploitation Linux ou Windows, le BIOS est normalement configuré pour activer le MBR de celui ci, les étapes du démarrage peuvent alors commencer ...



# Démarrage du système Linux

## ❖ Processus du démarrage :

### I. Bios



# Démarrage du système Linux

## ❖ Processus du démarrage :

### II. MBR

- Le Master Boot Record ou MBR est le nom donné au premier secteur adressable d'un disque dur (cylindre 0, tête 0 et secteur 1, ou secteur 0 en adressage logique) dans le cadre d'un partitionnement Intel.
- Il s'agit du boot primaire, la taille du MBR étant limitée à 512 octets, ce petit programme n'a pour fonction que de lancer le boot secondaire qui occupe un plus gros espace ailleurs sur le disque.



- Le Boot Secondaire a pour fonction d'activer le système d'exploitation, c'est à dire d'activer le noyau. Les boot primaire et secondaire constituent ce qu'on appelle le chargeur Grub

# Démarrage du système Linux

## ❖ Processus du démarrage :

### III. Chargeur de démarrage Grub2

- **GNU GRUB** (acronyme signifiant en anglais « **GR**and **Un**ified **Bo**otloader ») est un programme d'amorçage GNU qui gère la gestion du chargement des systèmes d'exploitation disponibles sur le système. Il permet à l'utilisateur de choisir quel système à démarrer. Il intervient après allumage de l'ordinateur et avant le chargement du système d'exploitation.
- **GRUB** dans sa version 2 (entièrement réécrite) est un chargeur de démarrage libre au même titre que Das U-Boot ou Barebox pour du matériel embarqué.



# Démarrage du système Linux

## ❖ Processus du démarrage :

### III. Chargeur de démarrage Grub2

#### a) Fichiers Grub2

La configuration de GRUB2 est composé de trois principaux fichiers :

- **/etc/default/grub** - le fichier contenant les paramètres du menu de GRUB 2,
- **/etc/grub.d/** - le répertoire contenant les scripts de création du menu GRUB 2, permettant notamment de personnaliser le menu de démarrage,
- **/boot/grub2/grub.cfg** - le fichier de configuration final de GRUB 2, non modifiable. (/boot/grub/grub.cfg sous Debian).

- Ce dernier fichier est généré automatiquement par le programme **grub2-mkconfig** à partir des scripts **/etc/default/grub** et **/etc/grub.d/** :



# Démarrage du système Linux

## ❖ Processus du démarrage :

### III. Chargeur de démarrage Grub2

#### a) Fichiers Grub2

Voici le contenu du fichier **/etc/default/grub** avec les principales variables d'environnement :

```
cat /etc/default/grub
```

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet"
GRUB_DISABLE_RECOVERY="true"
```

Sous Ubuntu pour générer le fichier de configuration de GRUB2 :

```
update-grub
```





# Démarrage du système Linux

## ❖ Processus du démarrage :

### III. Chargeur de démarrage Grub2

#### b) Gestion

Obtenir la version du noyau courant :

```
grub2-editenv list  
saved_entry=CentOS Linux (3.10.0-327.el7.x86_64) 7 (Core)
```

Pour connaître la liste des entrées du menu :

```
grep ^menu /boot/grub2/grub.cfg
```

Pour mettre à zéro Grub2

```
rm /etc/grub.d/*  
rm /etc/sysconfig/grub  
yum reinstall grub2-tools  
grub2-mkconfig -o /boot/grub2/grub.cfg
```

Pour réinstaller grub2

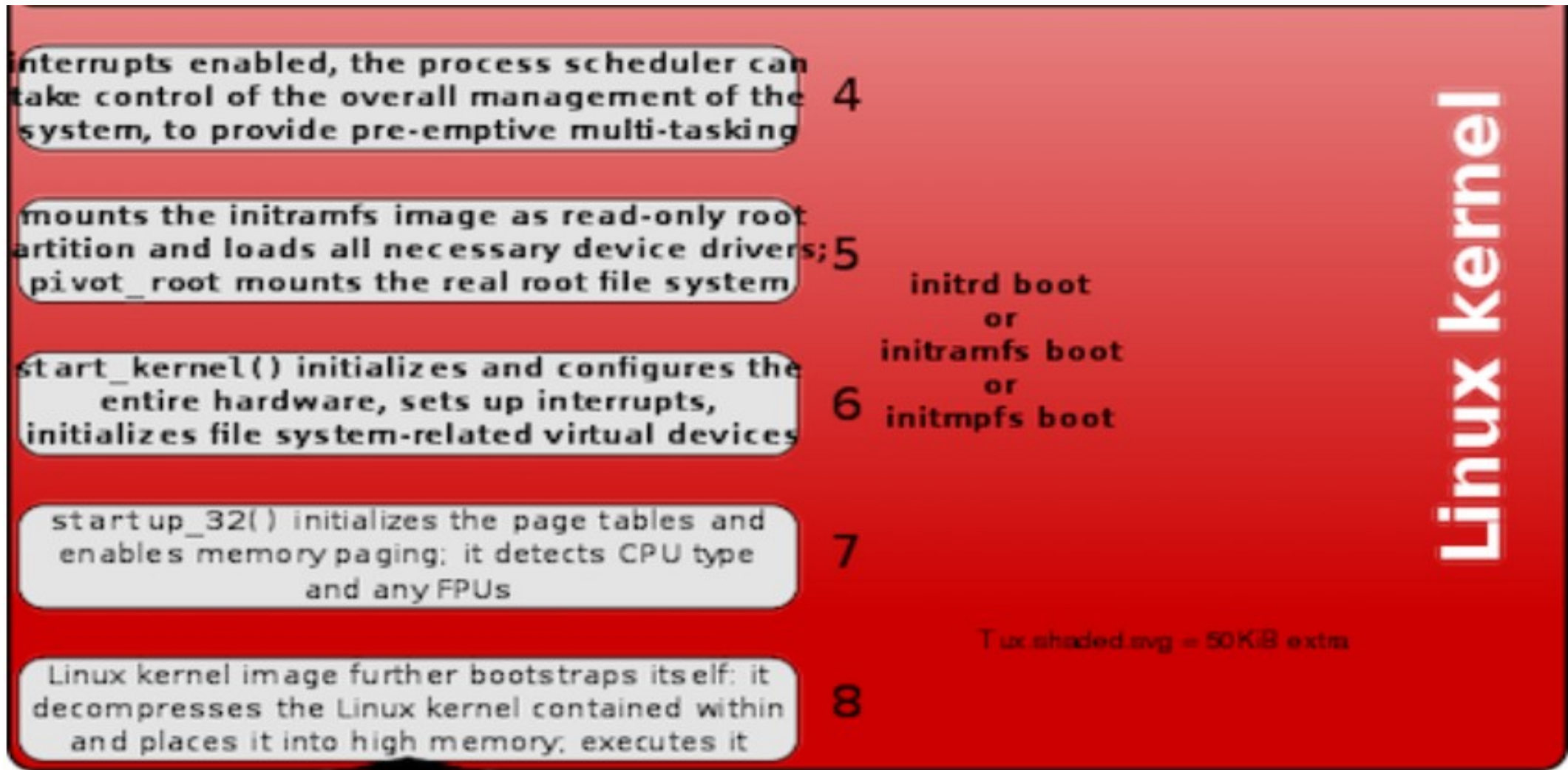
```
grub2-install /dev/sda1
```



# Démarrage du système Linux

## ❖ Processus du démarrage :

### IV. Linux kernel



# Démarrage du système Linux

## ❖ Processus du démarrage :

### V. init et systemd

La procédure de démarrage d'un ordinateur Linux peut se résumer de la manière suivante :

Le chargeur d'amorçage (GRUB2 a priori) charge le noyau, ensuite le noyau monte le système de fichier racine (le « / »), puis il initialise la console initiale :

❑ **init** (abréviation de “initialization”) est le programme sous Unix qui lance ensuite toutes les autres tâches (sous forme de scripts). Il s'exécute comme un démon informatique. Son identifiant de processus (PID) est 1.

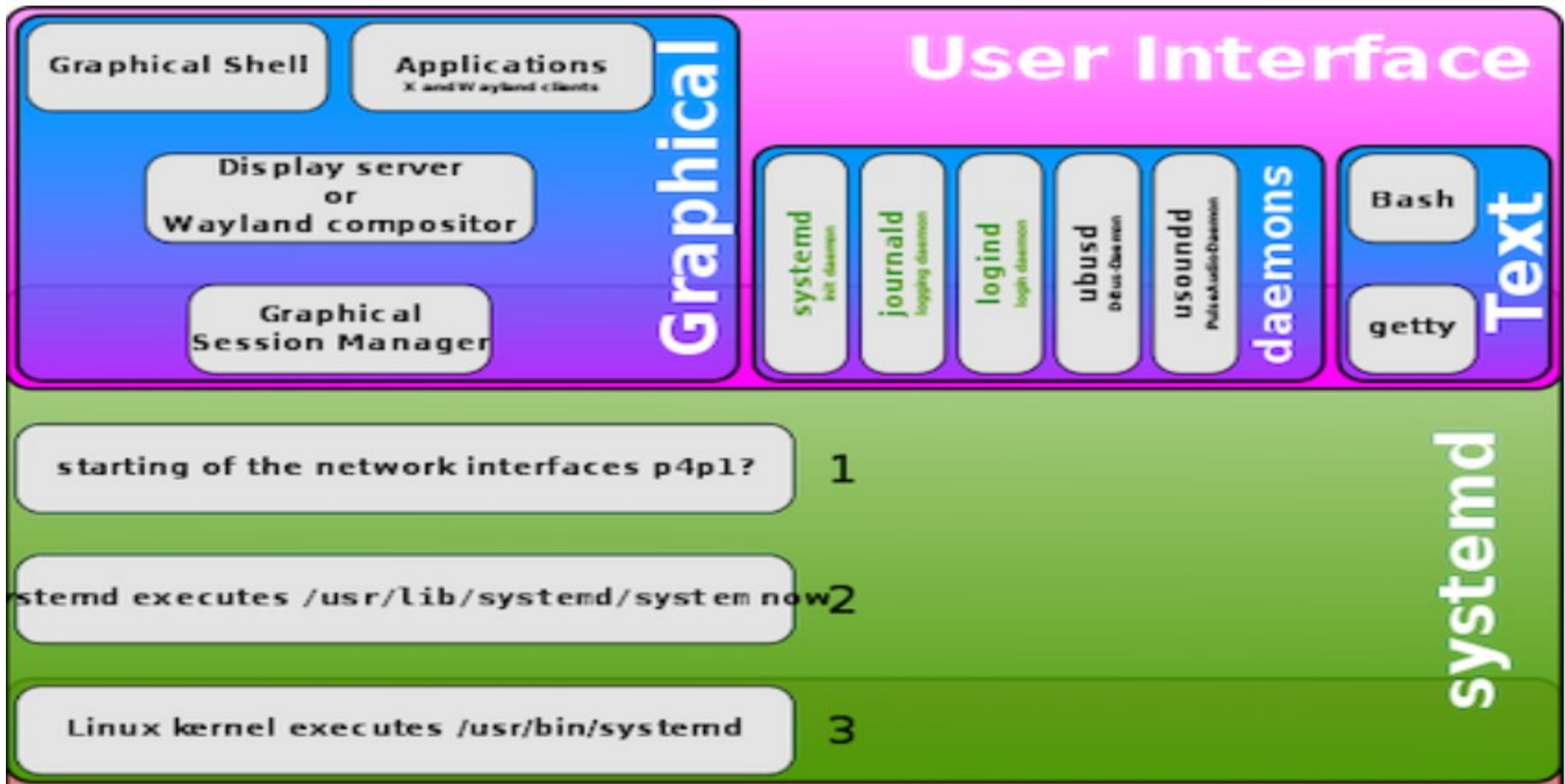
❑ **systemd** est une alternative au démon init de System V. Il est spécifiquement conçu pour le noyau Linux. Il a pour but d'offrir un meilleur cadre pour la gestion des dépendances entre services, de permettre le chargement en parallèle des services au démarrage, et de réduire les appels aux scripts shell.



# Démarrage du système Linux

## ❖ Processus du démarrage :

V. init et systemd



# Démarrage du système Linux

## ❖ Processus du démarrage :

### VI. Run levels

Le “run level”, ou niveau de fonctionnement, est un chiffre ou une lettre utilisé par le processus init des systèmes de type Unix pour déterminer les fonctions activées du système.

Dans cette organisation héritée de UNIX System V, les scripts de lancement des applications sont regroupés dans un répertoire commun `/etc/init.d`. Ces scripts reçoivent un paramètre qui peut être start, stop, restart, etc.

À chaque niveau correspond un répertoire (typiquement `/etc/rc.d/rc2.d` pour le niveau 2) de liens symboliques vers des fichiers de `/etc/init.d`. Ces liens symboliques portent des noms commençant par la lettre S ou K, suivi d'un numéro sur deux chiffres.



# Démarrage du système Linux

## ❖ Processus du démarrage :

### VI. Run levels

Lors d'un changement de run level :

- ❖ Les scripts dont le nom commence par un K dans le répertoire correspondant au niveau actuel sont lancés (dans l'ordre des numéros) avec le paramètre stop, ce qui a normalement pour effet d'arrêter le service correspondant,
- ❖ les scripts du nouveau niveau qui commencent par S sont appelés successivement avec le paramètre start.

Avec init, les niveaux d'exécutions servent à ces usages :

- ☐ Niveau 1. Mode mono-utilisateur ou maintenance
- ☐ Niveau 2. mode multi-utilisateur sans ressources réseaux (NFS, etc)
- ☐ Niveau 3. mode multi-utilisateur sans serveur graphique
- ☒ Niveau 5. mode multi-utilisateur avec serveur graphique
- ☐ Le niveau 0 arrête le système.
- ☐ Le niveau 6 redémarre le système.

Pour vérifier le niveau d'exécution

```
runlevel
```

```
N 5
```

Pour se placer dans un niveau d'exécution

```
init x
```



Sous Ubuntu, le Niveau 2 est le seul niveau fonctionnel avec réseau et serveur graphique. Les niveaux 3 et 5 ne sont pas utilisés.

# Démarrage du système Linux

## ❖ Processus du démarrage :

### VI. Run levels

Obtenir le niveau d'exécution par défaut :

```
systemctl get-default
```

```
graphical.target
```

Pour fixer le niveau d'exécution par défaut en mode multi-utilisateur avec serveur graphique :

```
systemctl set-default graphical.target
```

Pour passer mode maintenance avec un système de fichier local monté

```
systemctl rescue
```

Passer en mode maintenance avec seulement /root monté

```
systemctl emergency
```

Pour passer en mode multi-utilisateur sans serveur graphique (N 3)

```
systemctl isolate multi-user.target
```

Pour passer en mode multi-utilisateur avec serveur graphique (N 5)

```
systemctl isolate graphical.target
```





# Démarrage du système Linux

## ❖ Processus du démarrage :

### VI. La commande systemctl

C'est une commande dédiée au systemd. Le systemd permet de configurer les services qui sont lancés au démarrage.

- De Lister les services actifs par cette commande : « **systemctl list-units --type=service** »
- De Connaitre l'état d'un service par cette commande : « **systemctl is-active nom\_du\_service** »
- Activer / Désactiver un service au démarrage par cette commande :  
« **systemctl enable nom\_du\_service** » et « **systemctl disable nom\_du\_service** »
- Stop, start, restart and reload un service par cette commande :  
« **systemctl stop nom\_du\_service** » , « **systemctl start nom\_du\_service** » , « **systemctl restart nom\_du\_service** » et « **systemctl reload nom\_du\_service** »
- Lister les dépendances d'un service par cette commande :  
« **systemctl list-dependencies nom\_du\_service** »





# Démarrage du système Linux

## ❖ Démarrage, redémarrage et éteint un système Linux:

Sur une machine locale ou virtuelle ou un serveur distant.

Pour redémarrer le système, on peut procéder à ces différentes manières :

### Arrêter le système

```
systemctl halt  
shutdown -h now  
halt  
init 0
```

### Eteindre le système

```
poweroff  
systemctl poweroff
```

### redémarrer le système

```
systemctl reboot  
shutdown -r now  
reboot  
init 6
```

### Suspendre le système

```
systemctl suspend
```

### Hibernation

« pour enregistrer l'état de la machine »

```
systemctl hibernate
```

