# Introduction à l'Informatique

# Cours 2

kn@lmf.cnrs.fr
https://usr.lmf.cnrs.fr/~kn

université
PARIS-SACLAY

# Plan

# Définitions

Programme : séquences d'instructions effectuant une tâche sur un ordinateur.

Exécutable : fichier binaire contenant des instructions machines interprétables par le microprocesseur.

Processus : instance d'un programme (≡ « un programme en cours d'exécution »).

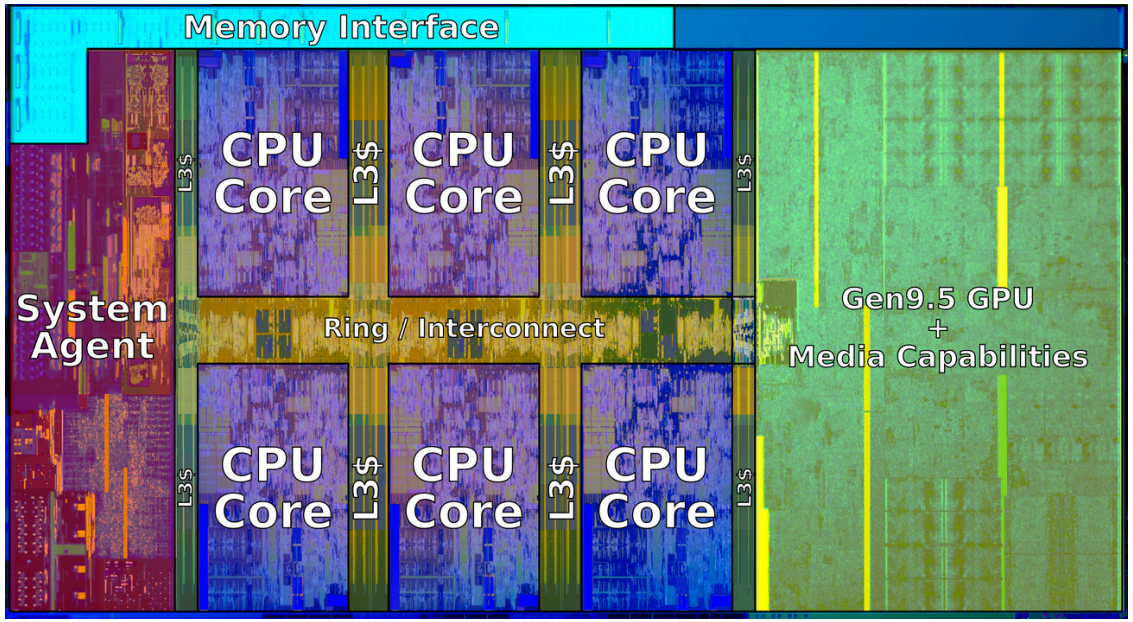
# Exécuter un programme ?

Pour exécuter un programme on peut

- Cliquer sur l'icône de l'exécutable, dans une interface graphique
- Entre le chemin vers le fichier exécutable dans le terminal (ou simplement son nom, si le fichier est dans un répertoire prédéfini).

Le programme s'exécute... oui mais comment ?

# Microprocesseur



# Modèle de Von Neumann

Proposé par John Von Neumann en 1945

Un processeur

- Une Unité Arithmétique et Logique (ALU) : calcul
- Un banc de registres : opérandes et résultats
- Une unité de contrôle : exécution

De la mémoire

Des périphériques d'entrée/sortie

# Jeu d'instructions

Un processeur est un circuit **programmable**. Il sait exécuter un certain nombre d'opérations (arithmétiques, logiques, lecture/écriture en mémoire, ...).

L'ensemble de ces instructions s'appelle le **jeu d'instruction**

Chaque instruction possède une représentation en binaire (suite de 0 et de 1)

Par exemple, sur les processeur Intel, **`01101001`** représente la multiplication

# Mémoire et registre

- La mémoire est un grand tableau d'octets (8 bits)
- L'indice d'une case dans le tableau s'appelle une adresse
- Un registre une petite case mémoire, située au sein du processeur sur laquelle le processeur peut travailler (additionner deux registres, faire la négation d'un registre, écrire un registre en mémoire, ...)

# Fonctionnement de l'architecture de Von Neumann

- Les instructions du programmes sont **stockées en mémoire**
- L'unité de contrôle possède un registre spécial : PC (*program counter*) qui indique l'adresse de la prochaine instruction à exécuter

1. L'unité de contrôle lit en mémoire la prochaine instruction et la décode (ex: **`01101001`** = multiplier)
2. Si besoin les opérandes sont stockées dans des registres
3. L'ALU effectue l'opération et sauve le résultat dans un nouveau registre (opération arithmétique)
4. Ou le processeur copie un registre dans la mémoire (opération d'écriture)
5. PC se positionne sur l'adresse de l'instruction suivante, et on recommence en 1

# Et donc quand on exécute un programme ?

Un fichier exécutable est un fichier contenant des instructions machines (comme `01101001`).

Lorsqu'on l'exécute :

1. Le système d'exploitation lit le fichier, il le copie dans une zone de la mémoire
2. Il copie l'adresse du début de la zone dans PC

# Plusieurs processus ?

Le modèle de Von Neumann seul ne permet pas d'exécuter « plusieurs programmes à la fois »

Les processeurs modernes possèdent aussi des alarmes configurables

Le système d'exploitation configure une alarme pour interrompre le processeur toutes les Xµs

Lors de cette **interruption** une portion de code spéciale est exécutée : le **gestionnaire de processus**

# Exemple

1. Exécution d'un jeu pendant 50µs
2. Interruption : le code spécial **sauvegarde les registres et l'état du processeur**, puis restaure les registres d'un autre programme (par exemple firefox)
3. Exécution de firefox pendant 50µs
4. Interruption : sauvegarde des registres puis, restauration de l'état d'un autre programme
5. Exécution du processus qui dessine le bureau pendant 50 µs
   ...
6. On remets dans les registre toutes les valeurs sauvées en 2, le jeu reprend comme si de rien n'était

C'est le *ordonanceur de processus* qui décide quel programme a la main et pour combien de temps (priorité aux tâches critiques par exemple)
Le système d'exploitation stocke pour chaque processus un ensemble d'informations, le PCB (*Process Control Block*).

# Proccess Control Block

Le PCB contient:

- l'*identificateur du processus* (pid)
- l'*état* du processus (en attente, en exécution, bloqué, ...)
- le compteur d'instructions (*i.e.* où on en est dans le programme)
- le *contexte courant* (état des registres, ...)
- position dans *la file d'attente de priorité globale*
- informations mémoire (zones allouées, zones accessibles, zones partagées)
- listes des fichiers ouverts (en lecture, en écriture), liste des connexions ouvertes, ...

...

# Opérations sur les processus



- *création* et *destruction* de processus
- *suspension* et *reprise*
- *duplication* (*fork*)
- modification de la *priorité*
- modification des *permissions*

# États d'un processus

Un processus change d'état au cours de son exécution

**Nouveau :** le processus est en cours de création

**Exécution :** le processus s'exécute

**En attente :** le processus attend un évènement particulier (saisie au clavier, écriture sur le disque, ...)

**Prêt :** le processus est prêt à reprendre son exécution et attend que l'OS lui rende la main

**terminé :** le processus a fini son exécution

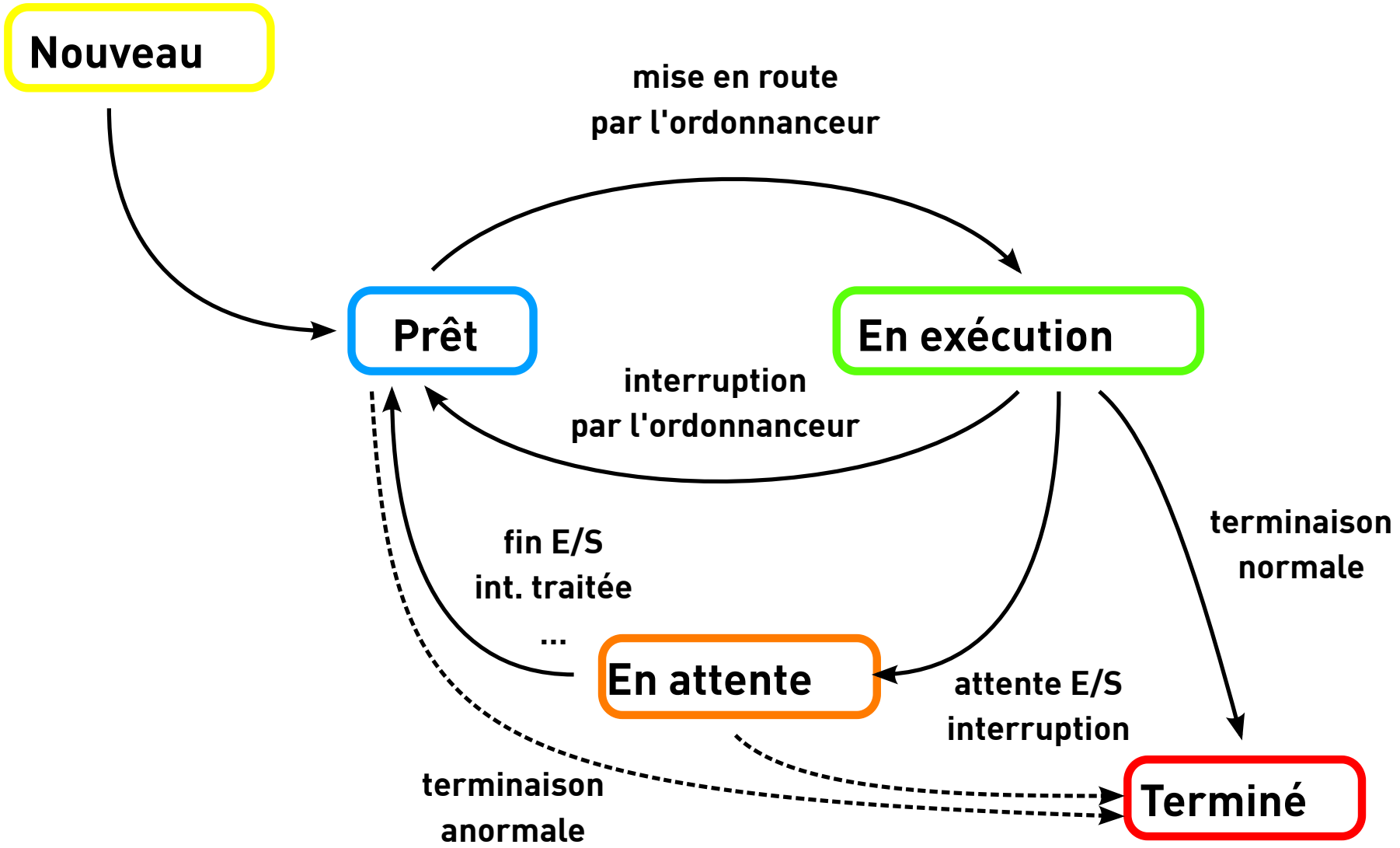
# États d'un processus

L'OS détermine et modifie l'état d'un processus:

- En fonction d'évènements internes au processus:
  - lecture d'un fichier (si le contenu n'est pas disponible, le processus passe de « prêt » à « en attente »)
  - le processus attends volontairement pendant x secondes

  ...
- En fonction d'évènements externes au processus:
  - un fichier devient disponible
  - un *timer* arrive à 0
  - le matériel déclenche une *interruption*

# États d'un processus



# La commande *ps*

Permet d'avoir des informations sur les processus en cours d'exécution (voir « **man ps** » pour les options):

```
$ ps -o user,pid,state,cmd x
          USER     PID   S CMD
          …
          kim      27030 Z [chrome] <defunct>
          kim      27072 S /opt/google/chrome/chrome --type=renderer
          kim      29146 S bash
          kim      29834 S evince
          kim      29858 S emacs cours.xhtml
          kim      29869 R ps -o user,pid,state,cmd x
```

# États des processus (sous Linux)



**R :** *Running* (en cours d'exécution)

**S :** *Interruptible sleep* (en attente, interruptible)

**D :** *Uninterruptible sleep* (en attente, non-interruptible)

**T :** *Stopped* (interrompu)

**Z :** *Zombie* (terminé mais toujours listé par le système)

# Signaux

L'OS peut envoyer des *signaux* à un processus. Sur réception d'un signal, un processus peut interrompre son comportement normal et exécuter son *gestionnaire de signal*. Quelques signaux:

| Nom | Code | Description |
|---|---|---|
| INT/TERM | 2,15 | demande au processus de se terminer |
| QUIT | 3 | interrompt le processus et produit un *dump* |
| KILL | 9 | interrompt le processus immédiatement |
| SEGV | 11 | signale au processus une erreur mémoire |
| STOP | 24 | suspend l'exécution du processus |
| CONT | 28 | reprend l'exécution d'un processus suspendu |

# Processus et terminal

Un processus est lié au *terminal* dans lequel il est lancé. Si on exécute un programme dans un terminal et que le processus ne rend pas la main, le terminal est bloqué

```
$ gedit
```

On peut envoyer au processus le signal *STOP* en tapant

```
ctrl-Z
```

dans le terminal:

```
$ gedit
^Z
[1]+  Stopped                 gedit
```

Le processus est suspendu, la fenêtre est gelée (ne répond plus).

# Processus et terminal

On peut reprendre l'exécution du programme de deux manières:

```
$ fg
```

Reprend l'exécution du processus et le remet en avant plan (terminal bloqué)

```
$ bg
```

Reprend l'exécution du processus et le remet en arrière plan (terminal libre)

On peut lancer un programme directement en arrière plan en faisant:

```
$ gedit &
```

On peut envoyer un signal à un processus avec la commande « **`kill [-signal] pid`** »

```
$ kill -9 2345
```

# Processus et entrées/sorties



Le terminal et le processus sont liés par trois fichiers spéciaux:

1. L'entrée standard (***stdin***), reliée au clavier
2. La sortie standard (***stdout***), reliée à l'affichage
3. La sortie d'erreur (***stderr***), reliée à l'affichage

Dans le *shell*, on peut utiliser les opérateurs **<**, **>** et **2>** pour récupérer le contenu de ***stdin***, ***stdout*** et ***stderr***:

```
$ sort < toto.txt
$ ls -l  > liste_fichiers.txt
$ ls -l * 2> erreurs.txt
```

# Shell et entrées/sorties



Dans le *shell*, l'opérateur | permet d'enchaîner la sortie d'un programme avec l'entrée d'un autre:

```
$ ls -l *.txt | sort -n -r -k 5 | head -n 1
```

1. affiche la liste détaillée des fichiers textes
2. trie (et affiche) l'entrée standard par ordre numérique décroissant selon le 5ème champ
3. affiche la première ligne de l'entrée standard

```
-rw-rw-r    1 kim kim 1048576 Sep 24 09:20 large.txt
```

# Fonctionnement des redirections

*cmd < fichier* :
: *fichier* est ouvert en lecture avant le lancement de *cmd*, le contenu est redirigé vers l'entrée standard de *cmd*.

*cmd > fichier* :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe il est tronqué à la taille 0. La sortie standard de *cmd* est redirigée vers *fichier*.

*cmd >> fichier* :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe, le curseur d'écriture est placé en fin de fichier. La sortie standard de *cmd* est redirigée vers *fichier*.

*cmd 2> fichier* :
: Comme `>` mais avec la sortie d'erreur

*cmd 2>> fichier* :
: Comme `>>` mais avec la sortie d'erreur

# Attention à l'ordre d'exécution !



Quelques exemples de commandes problématiques :

```
$ sort fichier.txt > fichier.txt
```

**fichier.txt** devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt > fichier.txt
```

**fichier.txt** devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt >> fichier.txt
```

**fichier.txt** contient son contenu original, suivi de son contenu trié !

```
$ cat < fichier.txt >> fichier.txt
```

**fichier.txt** est rempli jusqu'à saturation de l'espace disque !

# Quelques explications (1/2)



La commande `sort` doit trier son entrée standard. Elle doit donc la lire intégralement avant de produire la moindre sortie. Pour

```
$ sort < fichier.txt >> fichier.txt
```

on a donc :

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de toute l'entrée
4. Écriture de toute la sortie en fin de *fichier.txt*

# Quelques explications (2/2)



La commande **cat** ré-affiche son entrée standard sur sa sortie standard. Elle peut donc lire le fichier morceaux par morceaux et les afficher au fur et à mesure. Supposons que *fichier.txt* contient **AB** :

```
$ cat < fichier.txt >> fichier.txt
```

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de A (et positionnement du curseur de lecture sur B)
4. Écriture de A en fin de fichier *fichier.txt*
5. Lecture de B (et positionnement du curseur de lecture sur A)
6. Écriture de B en fin de fichier *fichier.txt*
7. Lecture de A (et positionnement du curseur de lecture sur B)
8. Écriture de A en fin de fichier *fichier.txt*
9. ...

# Conseils...



On évitera toujours de manipuler le même fichier en entrée et en sortie. Il vaut mieux rediriger vers un fichier temporaire, puis renommer ce dernier (avec la commande `mv`).

# Enchainement de commandes et code de sortie

Sous Unix, chaque commande renvoie un code de sortie (un entier entre 0 et 255).

Par convention, un code de 0 signifie terminaison normale, un code différent de 0 une erreur. On peut enchaîner des commandes de plusieurs façons :

*cmd*$_1$ ; *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$

*cmd*$_1$ && *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière réussit (code de sortie 0)

*cmd*$_1$ || *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière échoue (code de sortie différent de 0)

# La commande `test`

La commande **`test`** permet de tester des conditions sur les fichiers passés en arguments.

Si le test est vrai, la code de sortie est 0, sinon c'est 1

Les options permettent de spécifier les tests.

# La commande `test` (exemples)



L'option **`-f`** permet de tester si un fichier existe :

```
$ test -f toto.txt
```

La commande sort avec le code de sortie 0 si le fichier toto.txt existe et 1 sinon.
Comment s'en servir ?

```
$ test -f toto.txt && echo "Ok" || echo "Pas ok"
```

La commande affiche "Ok" si le fichier existe et "Pas ok" sinon.

# Plan

# Retour sur les exécutables

On a vu qu'un fichier exécutable est censé contenir des instructions machines.

Un humain ne peut pas écrire directement des instructions machines en binaire.

1. On peut utiliser un *compilateur* : Un programme qui transforme un fichier texte contenant le code source d'un programme (par exemple en C++) en instructions machines. (cf. les futurs cours d'Intro Prog Impérative).
2. On peut aussi utiliser un *interpréteur* : Un progamme qui lit des instructions dans un fichier texte et les exécutes.

- Le langage C++, utilisé en Intro. Prog. Impérative est *compilé*.
- Le langage du *shell* et la langage Python sont interprétés.

# Exécutables pour langages interprétés



Est-il possible d'écrire des programmes en *shell* ou en Python et d'en faire des fichiers exécutables ?

Oui !

On procède de la façon suivante

1. On crée un fichier **texte** (ex: **`test.sh`**) contenant le code dans le langage choisi (par exemple le shell)
2. On indique en début de fichier un commentaire spécial (appelé *shebang*) :

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

   Cette ligne commence par **`#!`** est suivie du chemin vers le programme qui interprète le langage (**bash**, **python3**, ...)
3. On rend ce fichier exécutable. On peut ensuite l'exécuter.

   ```
   $ chmod 755 test.sh
   $ ./test.sh
   ```

# Que fait le système d'exploitation ?



Lorsque l'on essaye d'exécuter une commande :

- Soit la commande est un fichier binaire, le système la copie en mémoire et configure le processeur pour qu'il exécute la première instruction.
- Soit le fichier commence par une ligne de la forme :

```
#!/chemin/vers/un/programme
...
texte
...
```

Le texte du fichier est copié sur l'entrée standard du programme dont le chemin est donné (« comme si un utilisateur avait saisi les lignes au clavier »)

# Script *Shell*

On peut utiliser cette fonctionalité pour écrire des *scripts shell*.

- Permet de combiner plusieurs commandes pour réaliser des traitements complexes
- Permet de stocker une fois pour toute une longue liste de commandes plutôt que de la saisir à chaque fois
- Comme dans tous les langages, on peut définir des variables
- On peut aussi utiliser des structures de contrôle avancées (boucles, tests, ...)

# Définitions de variables

On peut définir des variables au moyen de la notation
`VARIABLE=`*`contenu`*
et on peut utiliser la variable avec la notation **`$VARIABLE`**

- Attention, pas d'espace autour du **`=`**
- nom de variable en majuscule ou minuscule
- contenu est une chaine de caractères. Si elle contient des espaces, utiliser `" ... "`

exemple de définition :

```
i=123
j="Ma super chaine"
TOTO=titi
echo $TOTO
```

exemple d'utilisation: **`echo $j $i $TOTO`**
affiche « **`Ma super chaine 123 titi`** »

# Variables spéciales

Les variables `$1`, `$2`, ... contiennent les arguments passés au script sur la ligne de commande.

La variable `$0` contient le chemin vers le script en cours d'exécution

Attention, il est recommandé de toujours encadrer l'utilisation d'une variable par des guillemets "

# Conditionnelle

La syntaxe est :

```
if commande
then
     ...
else
     ...
fi
```

***commande*** est évaluée. Si elle se termine avec succès, la branche **`then`** est prise. Si elle se termine avec un code d'erreur, la branche **`else`** est prise. On peut utiliser la commande **`test`** qui permet de tester plusieurs conditions (existance d'un fichier, égalités de deux nombres, ...) et se termine par un succès si le teste est vrai et par un code d'erreur dans le cas contraire

# Exemple de script

On est dans le fichier **backup.sh** :

```
#!/bin/bash
INPUT="$1"
if test -f "$INPUT"
then
   cp "$INPUT" "$INPUT".bak
   echo "Sauvegarde du fichier $INPUT réussie"
else
   echo "Erreur, le fichier $INPUT n'existe pas"
fi
```

On suppose que le script possède les bonnes permissions, qu'il se trouve dans le répertoire courant, et qu'un unique fichier **doc.txt** se trouve dans ce même répertoire

```
$ ./backup.sh doc.txt
Sauvegarde du fichier doc.txt réussie
$ ls
backup.sh doc.txt doc.txt.bak
$ ./backup.sh toto.txt
Erreur, le fichier toto.txt n'existe pas
```

# Conclusion

- On s'arrête la pour les commandes de base du *shell*.
- Outil précieux qui permet de scripter plusieurs tâches d'administration système (renommer des fichiers en masse, exécuter une même action dans plusieurs répertoires, ...)
- Expose les concepts systèmes utile à la programmation en général (chemin, permission, utilisateurs, ...)