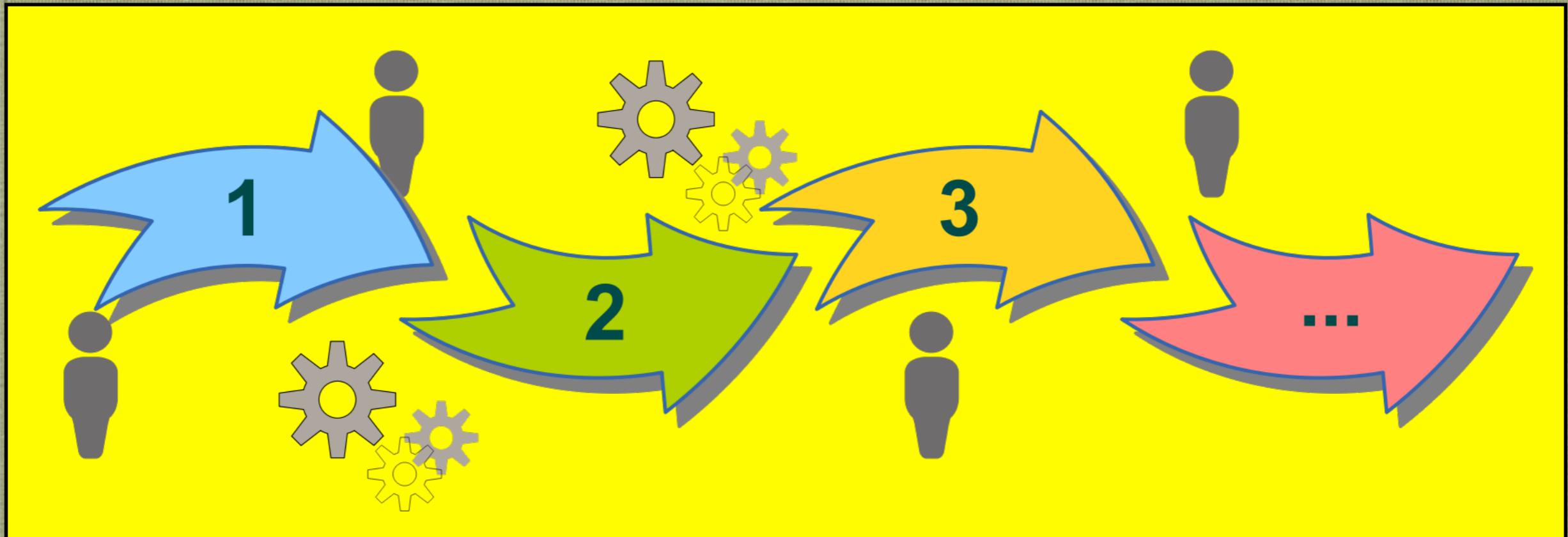


# Processus

- I. Aspects systèmes et commandes
2. Gérer des processus en python



## I) Aspects systèmes

# Les processus

Un **processus** est la forme logique sous laquelle le S.E. manipule un programme *en cours d'exécution* (aspect dynamique)

**A quoi ça sert-y donc c'truc là ?**

- à faire plusieurs activités en même temps : lire les nouvelles sur son navigateur pendant que le courrier se relève dans le gestionnaire de mail et qu'un EDI compile le projet qu'on doit rendre pour le lendemain.
- à permettre à plusieurs utilisateurs d'utiliser un même ordinateur en même temps : chacun peut utiliser des applications, quelles soient différentes ou déjà utilisées par d'autres utilisateurs.

# processus $\neq$ programme

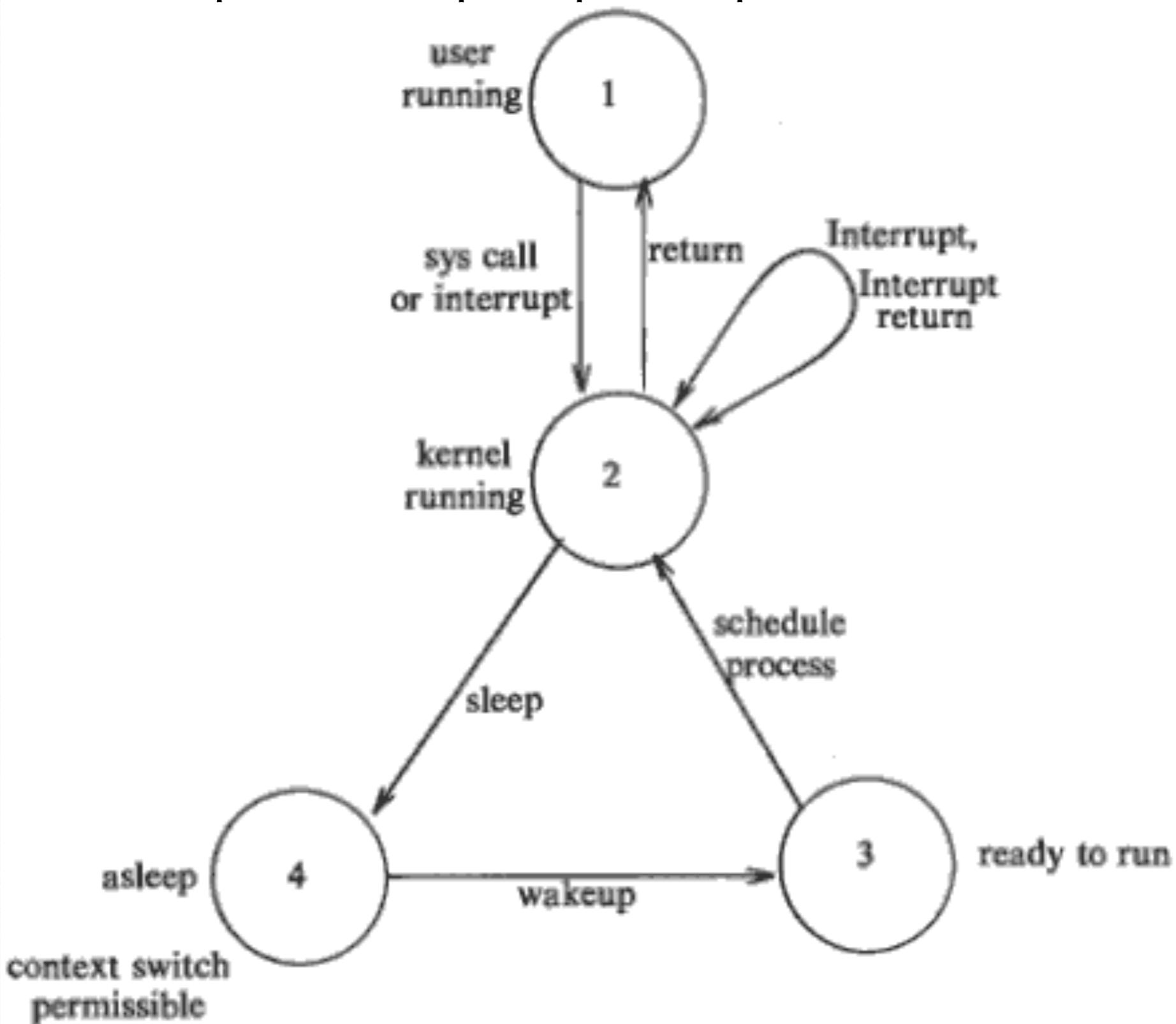


Ces deux notions ne  
sont pas équivalentes

- Il y a plus dans un processus que dans un programme :  
    suivant qui lance un processus, il accepte ou pas de se lancer, il donne ou pas le même résultat, ...
- Il y a plus dans un programme que dans un processus :
  - le programme **mozilla** utilise 3 à 4 processus
  - suivant le moment où vous lancez un programme, le nombre de processus créés diffère (*multi-threading*).  
Exemple = serveur web.

# Un processus dans tous ses états

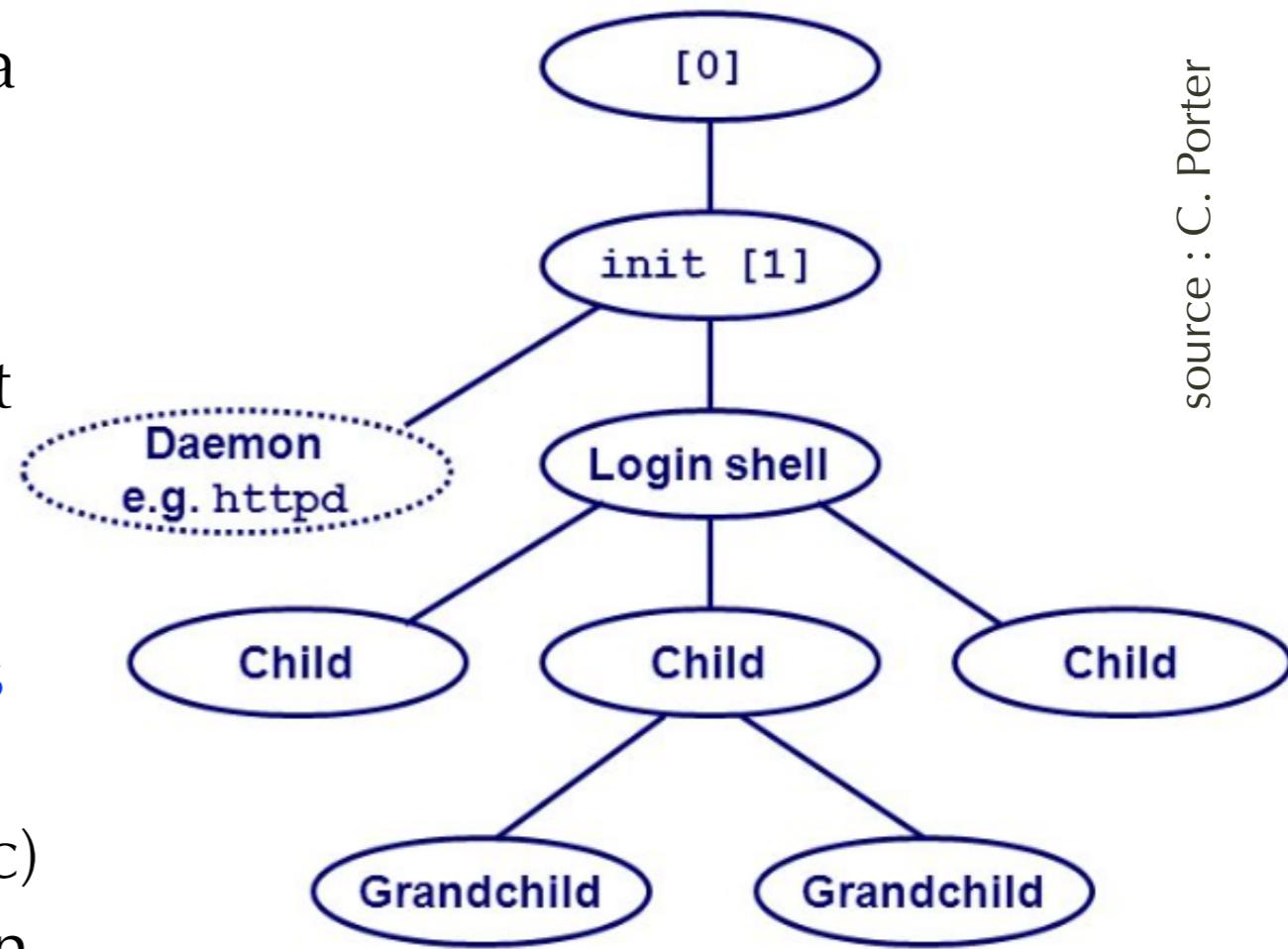
Lors de son existence (de son lancement au moment où il se termine), un processus peut passer par différents états :



# Organisation des processus

**Unix** : les processus sont gérés sous la forme d'une **hiérarchie** :

- Cette hiérarchie a une **racine**.
- Sur un Linux, le processus **init** est lancé par le noyau à la fin de son initialisation.
- À son tour, il crée des processus **fils** pour gérer les différents services du système (les *démons*, ex **httpd**, etc) dont un programme permettant à un utilisateurs de se connecter
- A la connexion une sous-arborescence de processus utilisateurs sont créés.
- quand un processus se termine, tous ses descendants aussi.





# Les processus



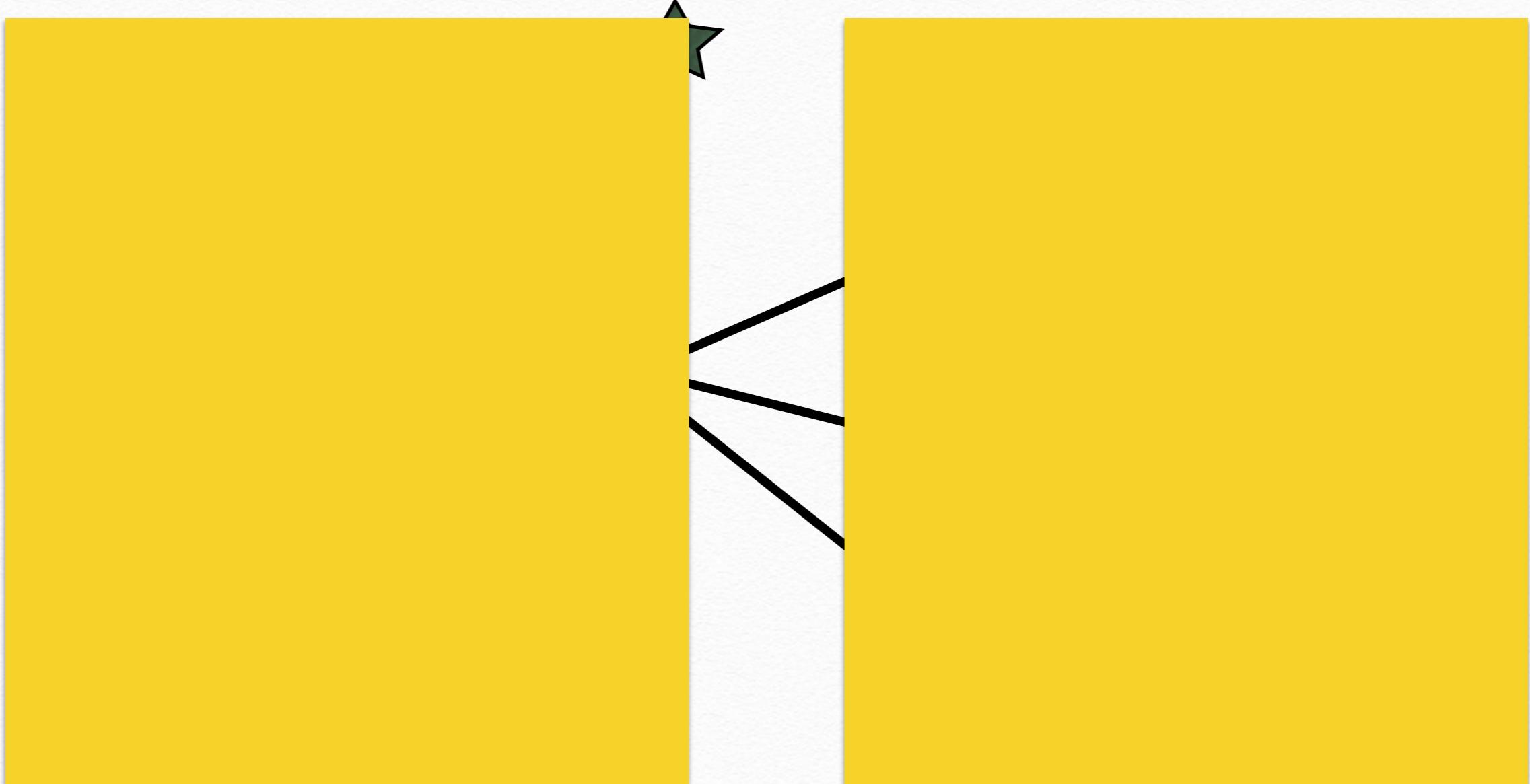
**A votre avis, quelles informations le système d'exploitation stocke-t-il pour un processus en cours d'exécution ?**



- The figure consists of five separate groups, each containing one large dark gray circle and three small black dots. The groups are arranged horizontally. The large circles are located at approximately (10, 10), (10, 30), (10, 50), (10, 70), and (10, 90). The small dots are located at approximately (25, 15), (35, 15), (45, 15) for the first group; (25, 35), (35, 35), (45, 35) for the second; (25, 55), (35, 55), (45, 55) for the third; (25, 75), (35, 75), (45, 75) for the fourth; and (25, 95), (35, 95), (45, 95) for the fifth.

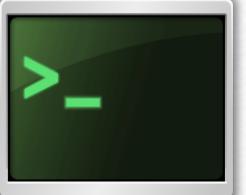
# Les processus

Tout processus a un **contexte** : l'ensemble des ressources utilisées par le programme pour pouvoir se dérouler



*Bloc de Contrôle  
d'un Processus*

Espace d'adressage en  
mémoire principale



# Commandes sur les processus

- Liste des processus : **ps**

[gaston:~] ps	temps CPU utilisé			
PID	TT	STAT	TIME	COMMAND
3899	pts/0	S	0:00.08	-zsh
4743	pts/0	S+	0:00.14	emacs
5190	?	S	0:00.04	mozilla

**numéro de processus**

**Pseudo terminal associé**

**état du processus:**

**temps CPU utilisé**

**commande exécutée**

R actif (*running*)  
T bloqué  
P en attente de page  
D en attente de disque  
S endormi (*sleeping*)  
IW swappé  
Z zombie

Question : que fait l'option -l à votre avis ? et -u ? et -x ?

# Liste des processus ... et encore plus

whereis  
top

D'autres commandes utiles :  
top, **pstree** et **htop**

Si l'une est manquante ? on l'installe en faisant  
appel à un gestionnaire de paquets

Raspbian, Ubuntu

apt-get install ....

Mac

Mac Ports

Homebrew

The missing package manager for OS X

Océmo



- Au fait à quoi sert la commande **sudo** ?

# Commandes sur les processus

- ♦ Tuer un processus :

`kill -9 <PID>`



- ♦ Certains processus «prennent la main» !

exemples : `find / Archibald -print > liste`  
ou : `emacs monFichier.txt`

- ♦ NB\* : Arrêter le processus lancé **en avant plan** dans un terminal : **CTRL-C**

- ♦ Meilleure solution : **&**

→ permet de lancer un processus **en arrière-plan** (c-a-d, sans bloquer le terminal)

*Note : la commande `open -a <nomApp> params` (Mac only)*

# Avant et arrière plan (suite)

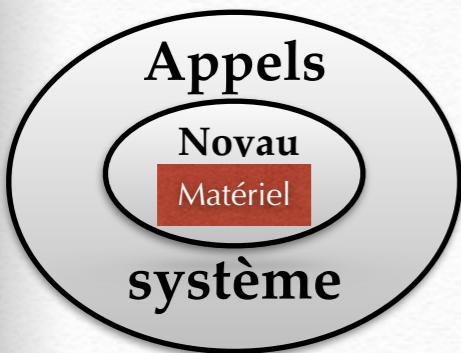


- ♦ **Interrompre** momentanément un processus lancé dans le terminal en ayant oublié d'indiquer le « & » : **CTRL-Z**
- ♦ **Relancer** un processus interrompu par **CTRL-Z** :
  - en avant plan : **fg**
  - en arrière plan : **bg**

# Notes sur l'exécution d'un processus



- Un processus peut avoir deux visages pour le CPU :
- **utilisateur** ou *user*, ou protégé : accès au matériel impossible, de même accès interdit à certaines zones mémoires et aux instructions privilégiés du noyau
- **noyau** ou *kernel*, ou « privilégié » : aucune restriction
- Changements entre processus en cours d'exécution dans le CPU sont gérés par **commutation de contexte**, suite à des « **interruptions** » matérielles ou logicielles (exceptions)

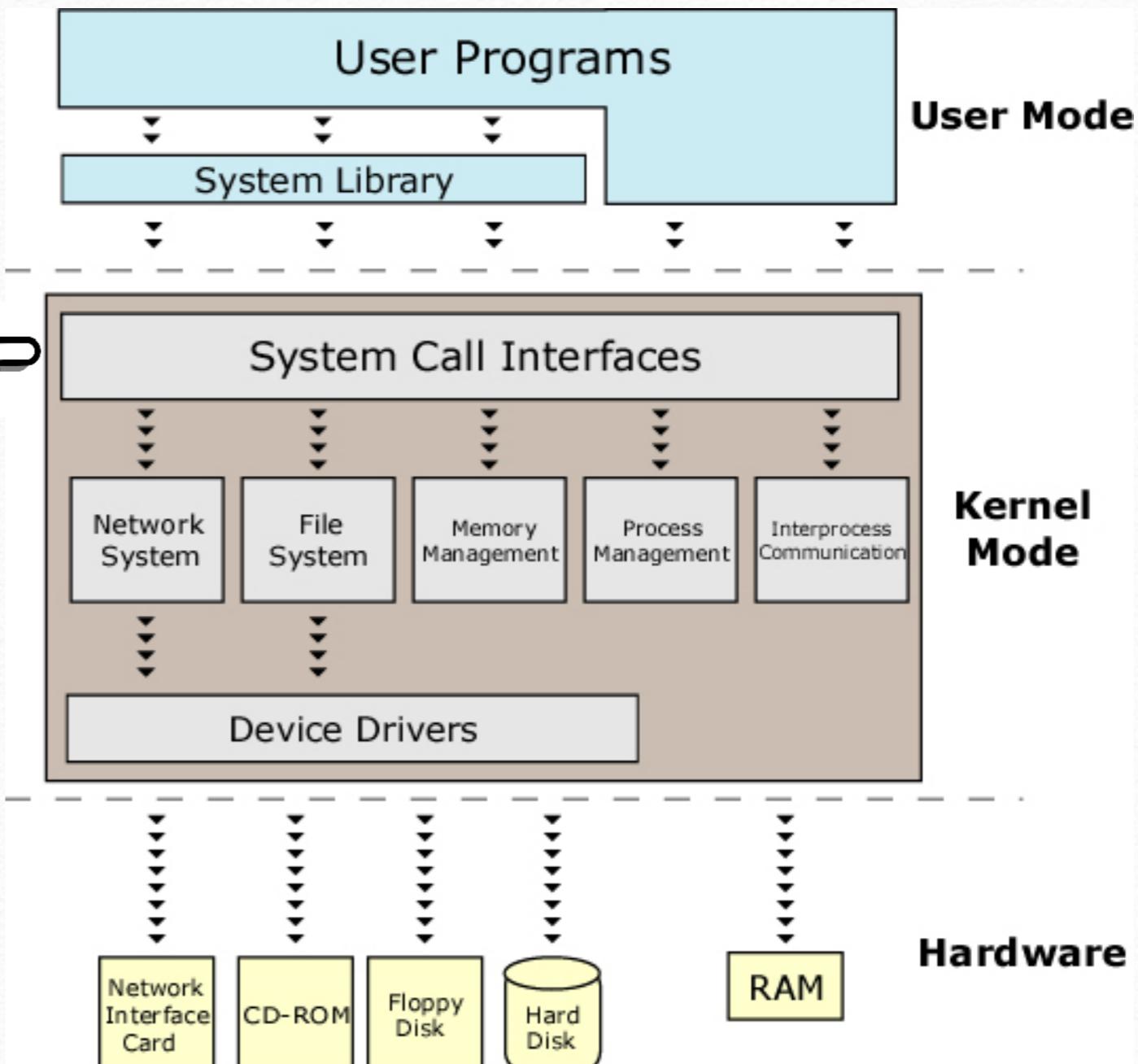


# Appels systèmes

- ▶ Ensemble de **services / fonctions** qu'un utilisateur peut demander au S.E. d'effectuer pour lui

**Exemples :**

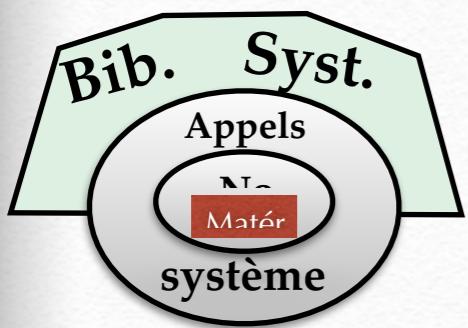
- .....
- ▶ La demande s'effectue par une commande shell ou par un programme utilisateur éventuellement par le biais d'une bibliothèque système
- ▶ L'appel système est réalisé en mode noyau (confiance)



**Pourquoi le S.E. fait confiance à un app.syst (kernel mode) ?**



**Pourquoi le S.E. ne fait pas confiance à un prog. utilisateur (user mode) ?**



# Bibliothèque système

Exemple : stdio.h (librairie standard d'entrées/sortie en C)

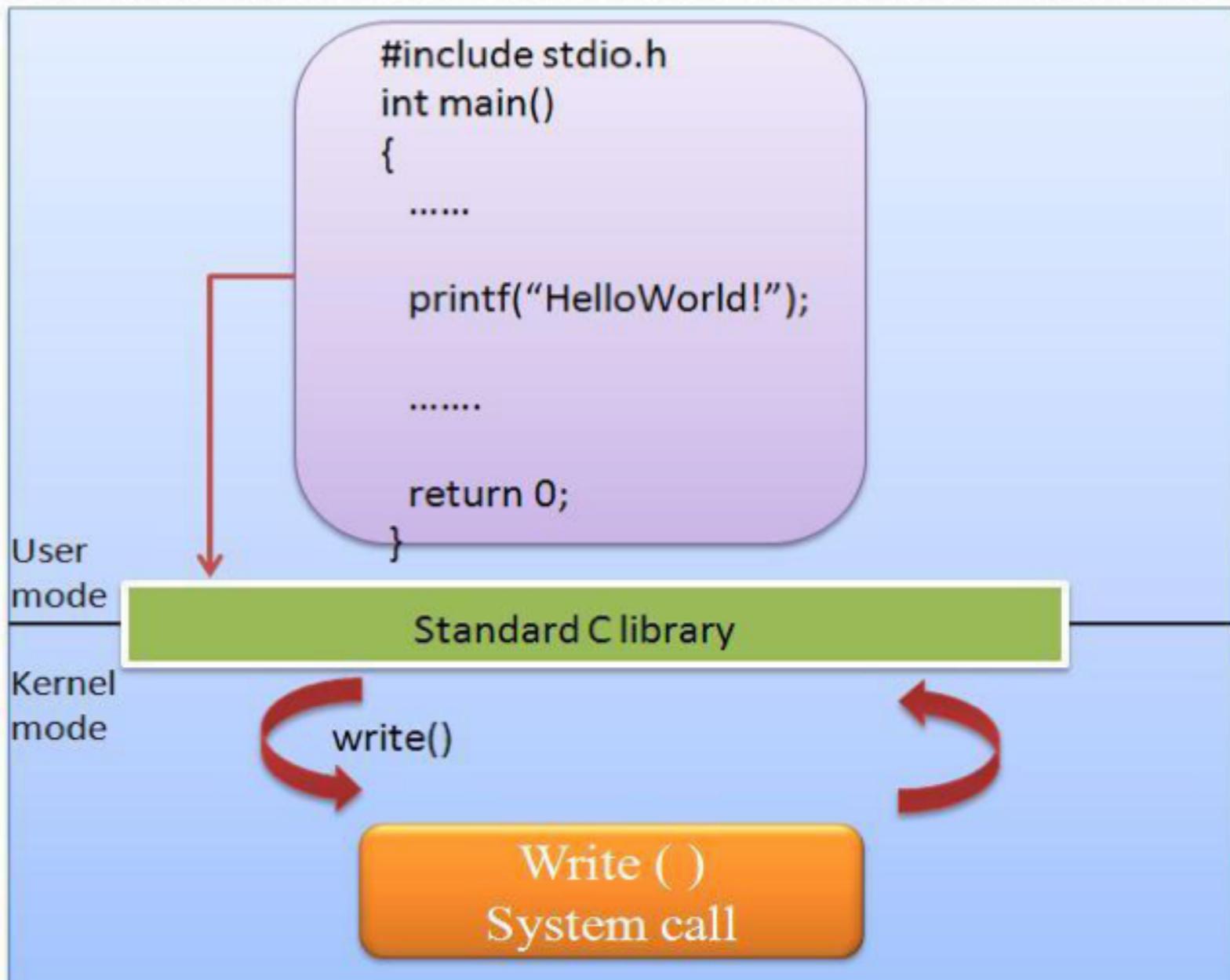
- ▶ Quand on écrit un prog., on ne réinvente pas tout : on s'appuie sur des bibliothèques de fonctions usuelles.

**Exemples :**

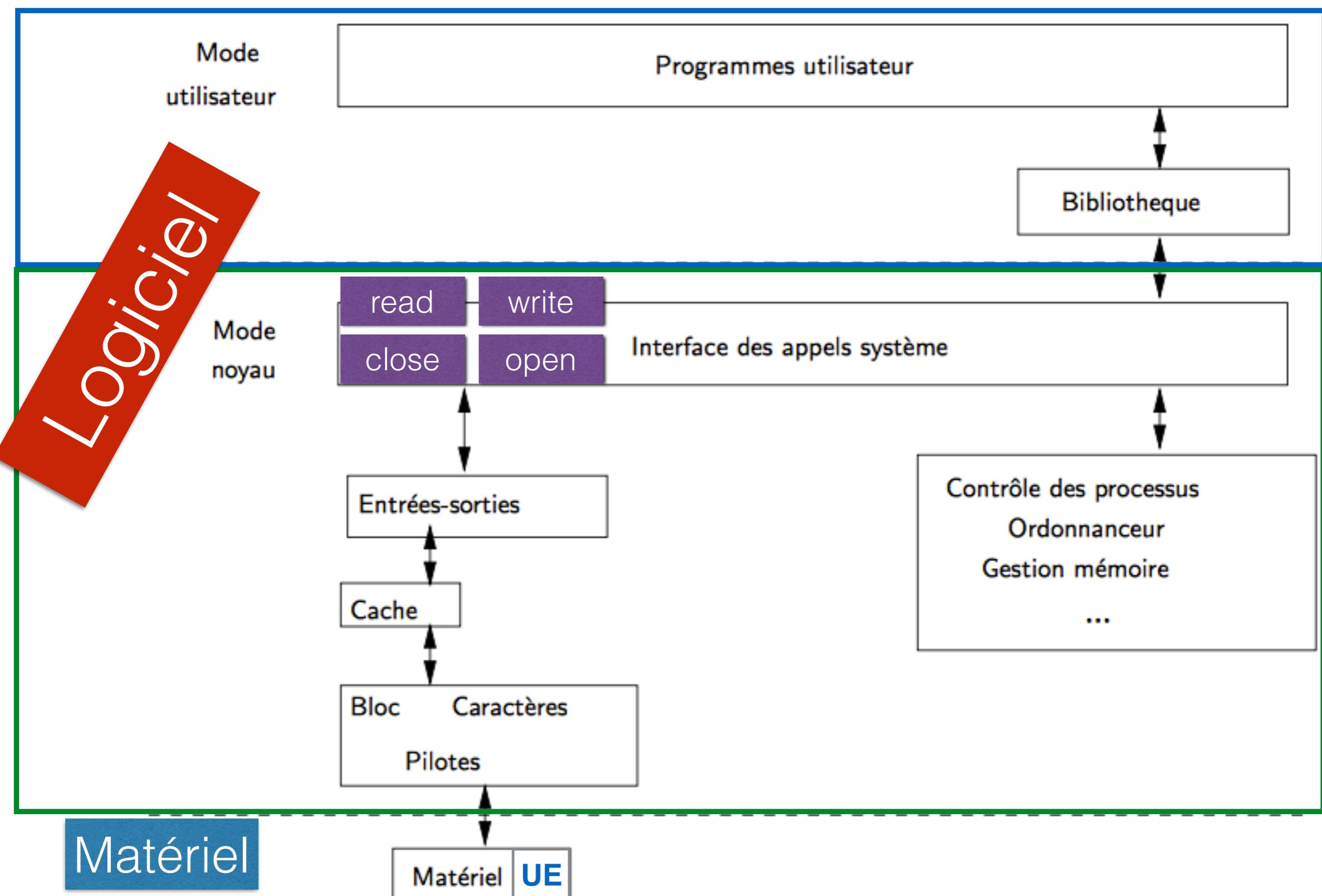
- .....
- .....

- ▶ Ses fonctions « enveloppent » chacune un appel système, pour rendre l'appel plus indépendant du système, du matériel.

**Intérêt :** .....



# Réalisation d'une E/S pour un programme



# Un exemple

## Programme en C

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return(0);
}
```

prog. C

bib. standard

Noyau

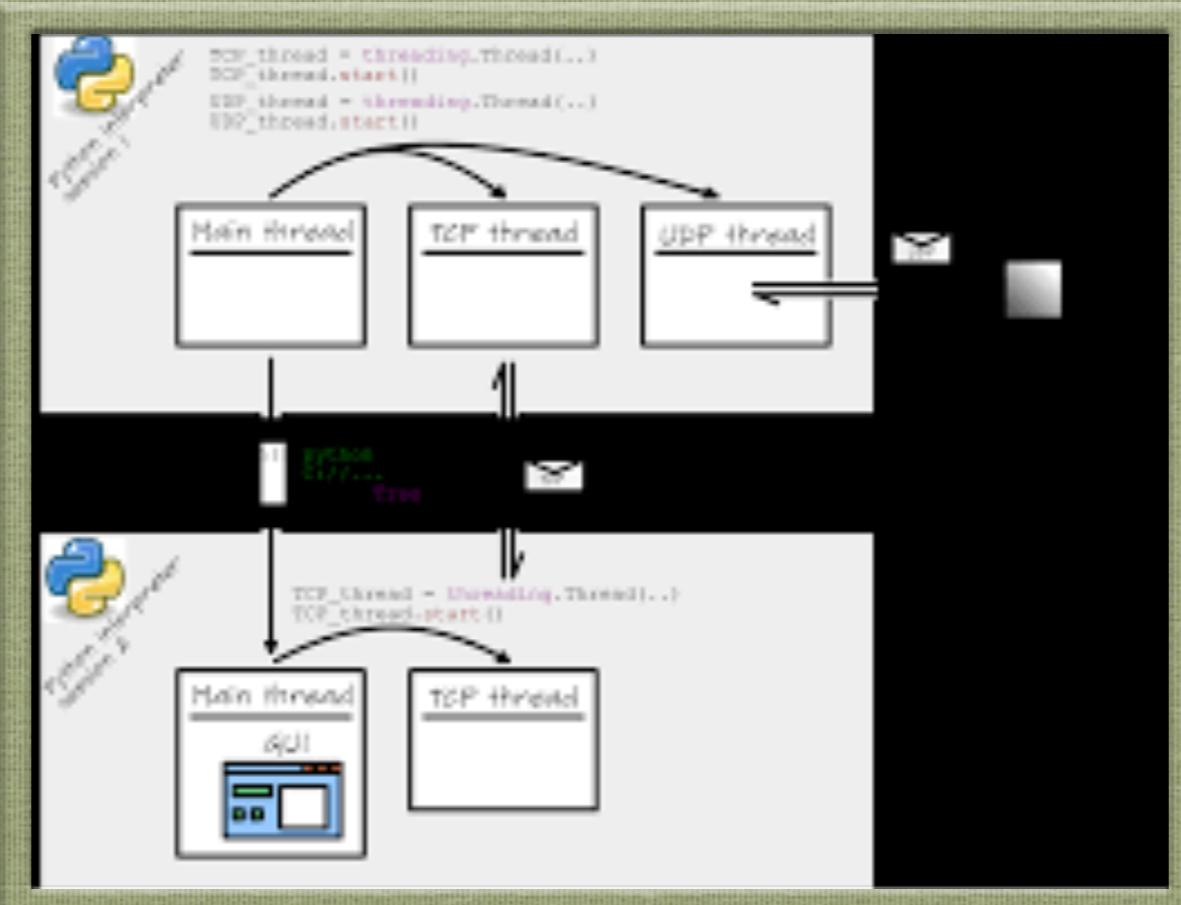
appel à printf()

instructions de printf()  
-> appel à write()

instructions de write()

## Code assemblé pour ce programme après compilation

```
.data                                # déclaration de section
msg:
    .string "Hello, world!\n"      # notre chaîne préférée
    len = . - msg                 # longueur de la chaîne
_start:
# Ecrit la chaîne dans la sortie écran standard (stdout)
    movl    $len,%edx    # 3ème argument: longueur du message
    movl    $msg,%ecx    # 2ème argument: pointe vers le msg
    movl    $1,%ebx      # 1er argument: desc.fich.(1 = stdout)
    movl    $4,%eax      # numéro de l'appel système (sys_write)
    int     $0x80        # fait un appel système
# et sort
    movl    $0,%ebx      # 1er argument: code de sortie, ici 0
    movl    $1,%eax      # numéro de l'appel système (sys_exit)
    int     $0x80        # fait un appel système
```



## II) Interaction avec les processus en python

# Objectifs

- Interagir avec le système Unix dans un script Python
- Exécuter des commandes & processus depuis un script
- Les signaux = une façon de communiquer entre processus

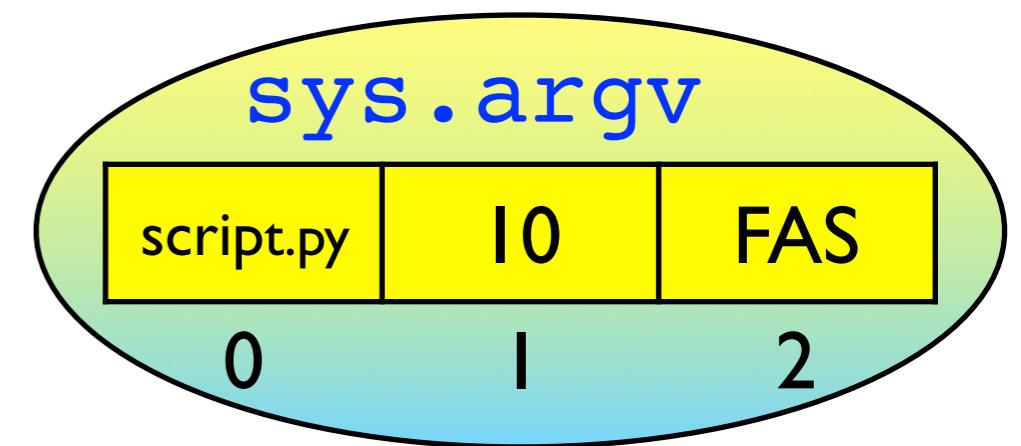


# Paramètres d'un script

- **sys.argv** : un tableau qui contient les arguments indiqués en ligne de commande par l'utilisateur au lancement du script

- Exemple : ./script.py 10 FAS

```
import sys  
  
for i, elt in enumerate(sys.argv):  
  
    print ("dans la case ",i," on trouve ",elt)
```



```
→ ('dans la case ', 0, ' on trouve ', 'script.py')  
( 'dans la case ', 1, ' on trouve ', '10')  
( 'dans la case ', 2, ' on trouve ', 'FAS')
```

Note : pour une gestion plus fine des paramètres on utilisera le module argparse

# Exécuter une commande shell depuis Python

```
import os  
os.system("ls -l")  
os.system("date > date.txt")
```

**Soucis :** on ne récupère pas un résultat éventuel de la commande (mais elle se déroule bien : le fichier date.txt existe après l'exécution de ce script).

# Exécuter une commande shell depuis Python ....et récupérer son résultat

```
>>> import os  
>>> res = os.popen("ls -l") ← en fait res est un tube («pipe») qui reçoit le résultat de la commande  
>>> res  
>>> <open file 'ls -l', mode 'r' at 0x1044a1660>  
  
>>> res.read()  
>>> 'total 56\n-rw-r--r-- 1 vberry staff 322 20  
mar 2017 AbstractTest.class\n-rw-r--r-- 1 vberry  
staff 363 20 mar 2017 AbstractTest.java\n-rw-r--r--  
1 vberry staff 363 20 mar 2017  
AbstractTest.java~\n-rwxr-xr-x 1 vberry staff 99  
8 oct 20:09 script.py\n-rw-r--r-- 1 vberry staff  
253 20 mar 2017 wrong.java~\n'
```



**Question :** comment obtenir un tableau de lignes de résultat plutôt qu'une seule chaîne de caractères ?

# Identification d'un processus



- Tout processus est identifié par un **pid**, et a un père (**ppid**) dans une hiérarchie des processus :

```
import os  
print "pid=%d" % os.getpid()
```

1. Ecrivez ce programme python

2. Si vous lancez plusieurs fois ce script, que remarquez vous ? Est-ce normal ?

3. Affichez maintenant le ppid du processus, remarquez-vous des changements ? Pourquoi ?



# Signaux

- C'est un moyen de communiquer une information à un processus
- Un script python peut traiter un signal qui lui est transmis :
  - enregistrer des données importantes avant de se terminer
  - libérer des ressources systèmes
  - etc

# Signaux en Python

- `import signal`
- Pour chaque signal pour lequel on veut effectuer un traitement spécifique, on déclare une fonction dédiée :

```
def fin_prog(signal, frame):  
    print("Allez on ferme !")  
    sys.exit(0)
```

référence de la fonction (pas de parenthèses)

puis dans le prog. principal, on assigne le signal à cette fonction :

```
signal.signal(signal.SIGINT, fin_prog)
```

# Signaux en Python



- Ecrivez un script `signaux.py` qui
  - vous demande un mot,
  - puis exécute une boucle infinie simulant un traitement long
  - a une fonction `fin_save` appelée quand il reçoit un signal de terminaison. Cette fonction crée un fichier `tmp.save` qui contient le mot reçu
- Vous lancerez le script dans un terminal, et constaterez d'abord qu'il ne se termine pas...
- Ensuite dans un autre terminal, vous obtiendrez le pid de ce processus par une commande Unix, puis chercherez à le tuer par la commande kill en lui passant les bons arguments (voir man) pour déclencher l'exécution de la fonction `fin_save`
- Vérifiez ensuite que le programme a bien créé le fichier comme attendu
- Attention : lors des tests pensez à supprimer le fichier créé ou à changer de mot d'un test sur l'autre

# Exécution en temps limité



- Modifiez le programme précédent de sorte que le programme s'arrête automatiquement au bout de 30s, même si l'utilisateur n'a rien saisi à la demande du programme.
- Indices :
  - Utiliser la fonction python `alarm()` et le signal **SIGALRM**
  - Il s'agit d'un signal, donc préparer une fonction spécifique (qui indique qu'il est temps de finir)
- Lors des tests vérifiez dans un autre terminal si la hiérarchie des processus est quand même bien créée



# Lancement de processus fils

- Parfois on a besoin qu'un script lance d'autres scripts (et non plus simplement des commandes systèmes) et que ces processus se synchronisent pour réaliser une tâche commune
- On dispose pour ça de plusieurs mécanismes :
  - ❖ processus lourds (**fork**, voir diapos suivantes), chacun ayant sa zone de mémoire propre
  - ❖ processus légers (**thread**, 2nd semestre) : partage d'une mémoire commune (attention!)

# Lancement de processus fils

```
#!/usr/bin/python
import os,sys
for i in range(3):
    pid = os.fork()
    if pid == 0:      # Le i-eme fils...
        sys.exit(i)   # Zombie tant pere ne fait pas wait/waitpid
    else:             # Le pere
        print "Je viens de creer le fils %d" % (pid)
# Le pere
rep = raw_input("blah...")
for i in range(3):
    pid, status = os.wait() ←
    print "Mort du fils %d" % (pid)
```



le fils devient zombie : un processus dont le père n'attend pas la mort

le zombie disparait

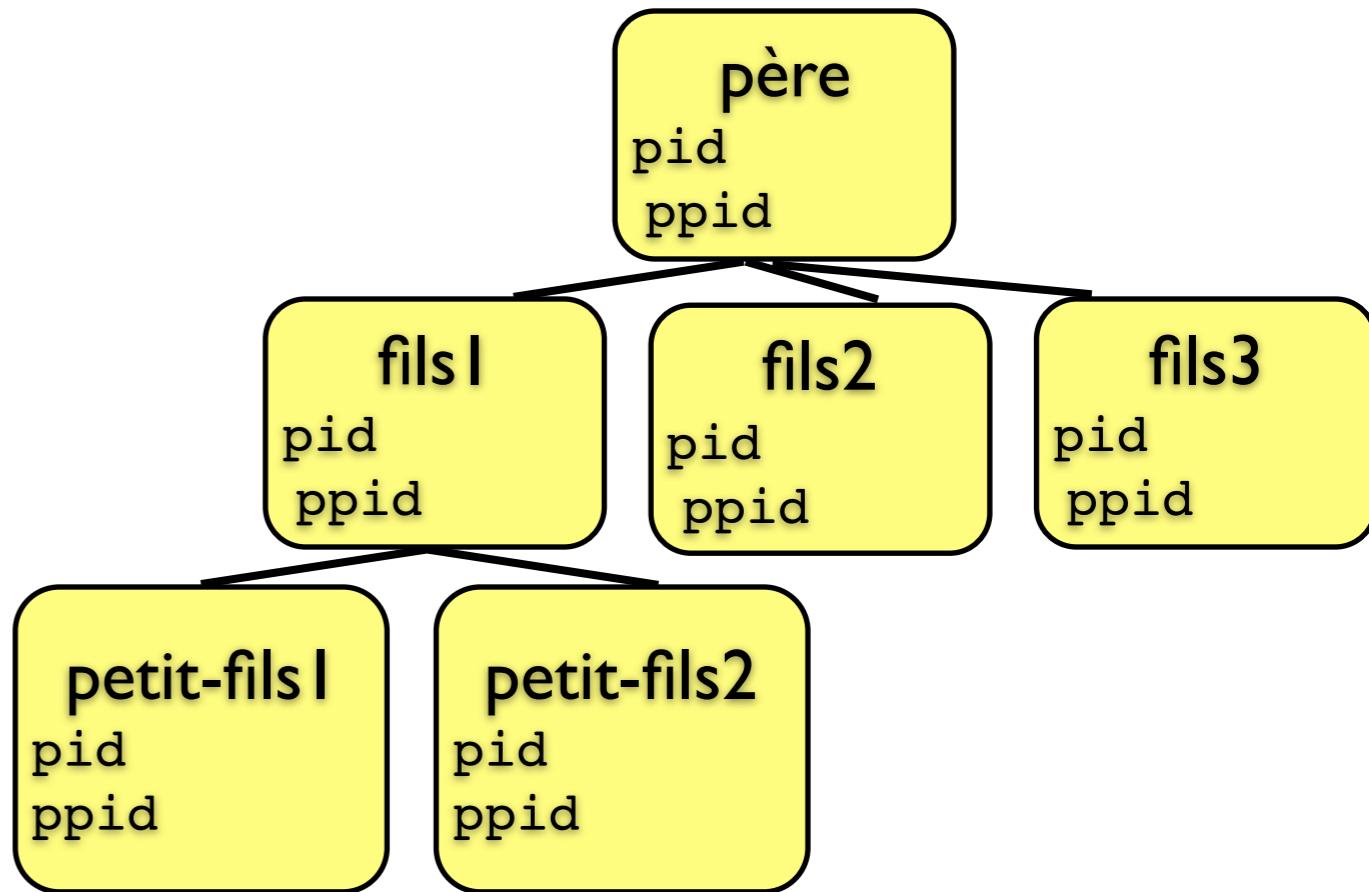


Lancez ce script dans un terminal, et dans un autre terminal utilisez la commande ps (éventuellement avec options :man) pour constatez l'apparition et la disparition de zombies

# Famille de processus



- Créez un script qui génère la hiérarchie de processus suivante :

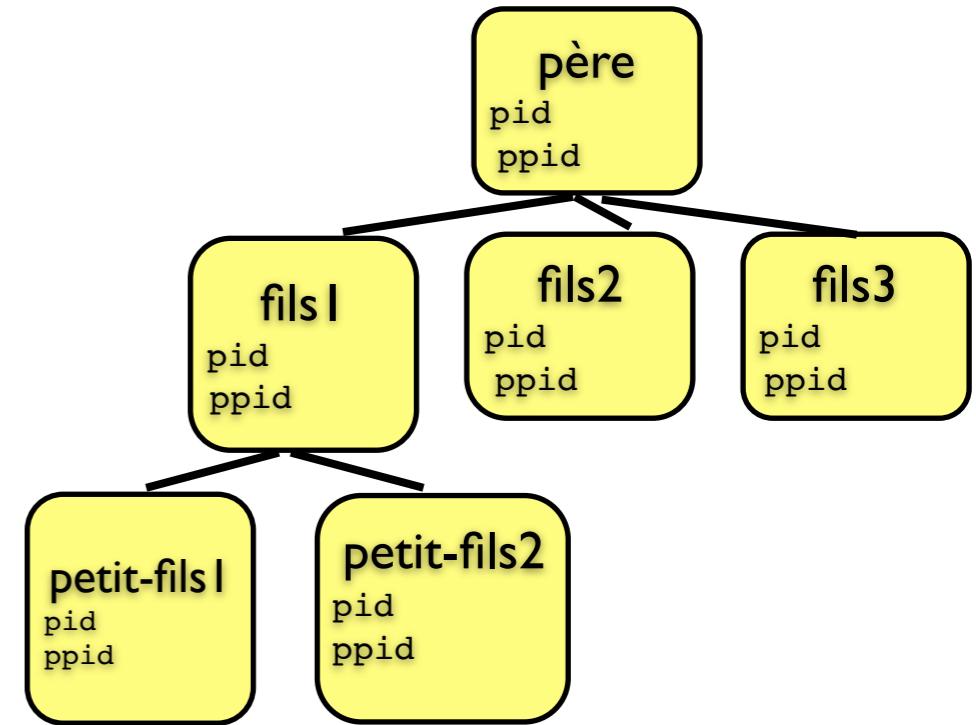


- le script possèdera une fonction `print_ids(mesg)` qui affichera qui il est (mesg), son pid et son ppid

# Famille de processus



- Pouvez-vous expliquer l'ordre dans lequel les lignes apparaissent à l'exécution ?
- Lancez plusieurs exécutions. Le résultat à l'écran est-il toujours dans le même ordre ? Pourquoi ?
- Modifiez votre programme pour que la génération la plus jeune émette en premier ses messages, puis la génération intermédiaire et enfin le processus initial (génération la plus ancienne)



# Recherche en famille



- Ecrire un script « `find_parallel` » tel que « `find_parallel.py fic rep` » effectue la recherche du fichier `fic` dans le dossier `rep`.
- Pour faire cette recherche, le script lance un fils pour s'occuper de chaque sous-répertoire dans le répertoire `rep`. Pour cela, chaque processus parcourt le contenu du répertoire dont le chemin lui est fourni ; s'il trouve un fichier de nom `fic`, il en affiche le chemin complet ; et pour chaque sous-répertoire, il crée un processus fils chargé d'y poursuivre la recherche.