In [45]:

```
Ce cours a été régénéré le 2019-11-08 01:31:45.583052. Mode sans corrigé. Mode statique.
```

Les processus

Les types de programmes

Les instructions qui sont exécutées sur l'ordinateur sont toutes écrites dans le même langage, qui est celui qui est compris par le processeur : le langage machine. Ce langage est composé de nombres, qui sont ensuite traduits en action à faire au niveau électronique du processeur, permet de contrôler l'action du processeur sur ses composants : déplacement de valeurs entre registres, communication avec la mémoire principale, communications basiques avec le reste des périphériques (souvent en modifiant des adresses mémoires réservées), opérations arithmétiques et logiques, changement de l'adresse d'exécution (avec ou sans conditions)...

Ce langage est directement compréhensible par l'ordinateur, mais ne l'est pas par les humains (il est éventuellement déchiffrable, mais c'est pénible). Les humains, pour programmer, utilisent donc d'autres langages, plus symboliques.

Les programmes compilés

Le premier de ces langages est le *langage assembleur (https://fr.wikipedia.org/wiki/Assembleur)*. Au lieu de décrire par des nombres les actions sur le processeur, il le décrit par des « verbes » élémentaires, inspirés de l'anglais et des notations qui permettent de décrire comment on prépare les données. Ce langage est en traduction directe vers l'assembleur : il n'y a pas un concept disponible en *assembleur* qui ne soit pas traduit de façon automatique par plus de 2-3 instructions en *langage machine*.

```
.text
.global _start

_start:
movl $4, %eax ; $4 = appel système write
movl $1, %ebx ; $1 = descripteur de fichier 1
movl $str, %ecx ; $str = adresse de la chaîne
movl $8, %edx ; $8 = longueur
int $0x80     ; appel système ($0x80)
movl $1, %eax ; $1 = appel système exit
movl $0, %ebx ; $0 = tout va bien
int $0x80     ; appel système ($0x80)

.data
str:
 .ascii "Bonjour\n"
```

La traduction en français de ce programme est : « mettre dans les registres `eax`, `ebx`, `ecx` et `edx` les valeurs 4, 1, l'adresse `str` (qui correspond au début d'une chaîne qui contient "Bonjour" avec retour à la ligne et caractère nul de terminaison) et 8. Ensuite le `int 0x80` dit de déclencher l'interruption numéro 80, qui correspond (sous Linux) au fait de faire un appel système. Le processeur exécute alors un appel système (il se trouve que pour eax=4, c'est l'appel système `write` qui va donc écrire sur le descripteur de fichier numéro 1 (parce que ebx vaut 1 — c'est donc la sortie STDOUT) une série de 8 caractères (parce que edx vaut 8) stockés à l'adresse `str` (parce que ecx vaut `str`). Les trois dernières lignes font de même, en invoquant cette fois l'appel système 1, comme on peut le lire dans cette table (https://syscalls.kernelgrok.com/), qui est l'appel `exit` (dont l'effet est exactement ce qu'on suppose : quitter le programme).

Comme on peut le constater, cette syntaxe est plus facile à comprendre que sa traduction numérique, mais encore ardue. Un spécialiste peut le faire, mais pas le développeur qui a d'autres préoccupations.

In [2]:

```
u.activite("Compilation d'un programme en assembleur (très facultatif)")
u.mark(r'''
Tapez dans un terminal le programme suivant :

    echo '
    .text
    .global _start

    _start:
    movl $4, %eax ; $4 = appel système write
    movl $1, %ebx ; $1 = descripteur de fichier 1
    movl $str, %ecx ; $str = adresse de la chaîne
    movl $8, %edx ; $8 = longueur
    int $0x80     ; appel système ($0x80)
    movl $1, %eax ; $1 = appel système exit
    movl $0, %ebx ; $0 = tout va bien
    int $0x80     ; appel système ($0x80)

    .data
    str:
     .ascii "Bonjour\n"

    '| sed -e 's/;.*$//g' > /tmp/monasm.s

    as -o /tmp/monasm.o /tmp/monasm.s
    ld /tmp/monasm.o -o /tmp/monasm
    # gcc --verbose -nostdlib -m32 -o /tmp/monasm /tmp/monasm.s
    /tmp/monasm
''')
```

Activité : Compilation d'un programme en assembleur (très facultatif)

Tapez dans un terminal le programme suivant :

```
echo '
.text
.global _start

_start:
movl $4, %eax ; $4 = appel système write
movl $1, %ebx ; $1 = descripteur de fichier 1
movl $str, %ecx ; $str = adresse de la chaîne
movl $8, %edx ; $8 = longueur
int $0x80     ; appel système ($0x80)
movl $1, %eax ; $1 = appel système exit
movl $0, %ebx ; $0 = tout va bien
int $0x80     ; appel système ($0x80)

.data
str:
 .ascii "Bonjour\n"

'| sed -e 's/;.*$//g' > /tmp/monasm.s

as -o /tmp/monasm.o /tmp/monasm.s
ld /tmp/monasm.o -o /tmp/monasm
# gcc --verbose -nostdlib -m32 -o /tmp/monasm /tmp/monasm.s
/tmp/monasm
```

On utilise donc pour la plupart des développements non pas l'assembleur, mais un langage de haut niveau qui permet de faire la même chose de façon plus lisible.

```
#include <unistd.h>

char *str="Bonjour\n";

int main() {
    write(1,str,8);
    return(0);
}
```

Ce programme est rédigé en **langage C** et comprend des aspects plus familiers : on peut utiliser des noms de variable, des types de données, des fonctions, la possibilité d'avoir des bibliothèques de fonctions pré-programmées... La fonction `main` a un rôle particulier d'être la fonction appellée au démarrage du programme et la valeur qu'elle retourne est renvoyée par l'appel système `exit` à la fin. Ce programme est la traduction du programme en assembleur présenté plus haut. Ou plus précisément, le programme va lui être traduit en assembleur (qui sera ensuite traduit lui-même en langage machine) avant d'être exécuté.

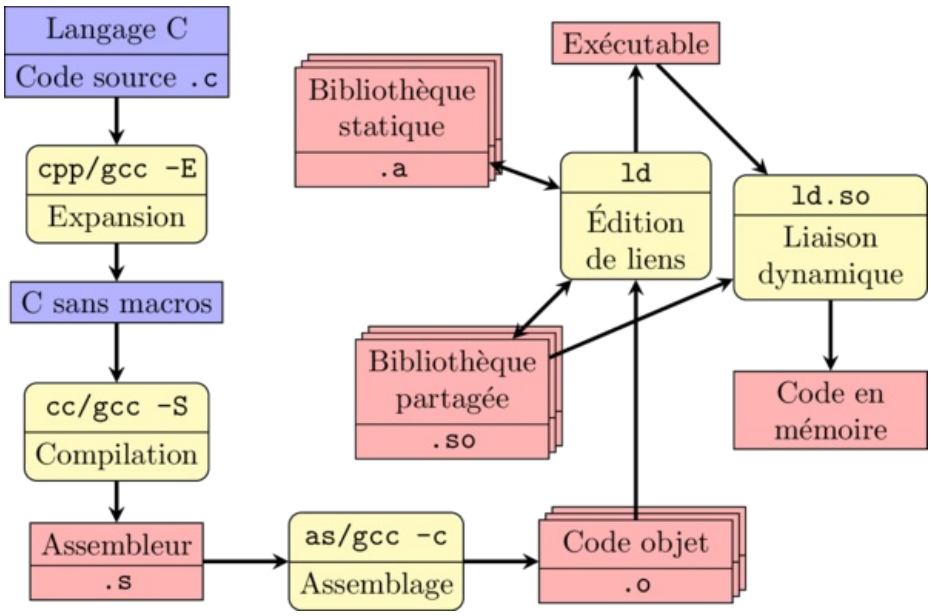
La possibilité d'avoir des *bibliothèques* de fonctions pré-programmées permet d'enrichir énormément un langage : par exemple, un programmeur C normal utiliserait plutôt le code suivant :

```
#include <stdio.h>

char *str="Bonjour\n";

int main() {
    printf(str);
    return(0);
}
```

Mais l'utilisation de bibliothèques de fonctions complique souvent la chaîne de fabrication d'un programme. En particulier, les fonctions pré-programmées, il faut les ajouter. Et il est possible de les ajouter directement, ou d'attendre le moment de l'exécution pour les ajouter, ce qui amène à un schéma plus complexe. L'ajout de bibliothèques de fonctions (le programme n'étant lui-même qu'une bibliothèque de fonction qui comprend la fonction `main`) suit donc le schéma ci-dessous.

In [3]:

Figure `compilation1*`.



In [4]:

Activité : La compilation

Y a-t-il des avantages à programmer directement en langage machine ? Lesquels ?

Votre proposition > _____

Y a-t-il des avantages à programmer directement en assembleur ? Lesquels ?

Votre proposition > _____

Y a-t-il des avantages à programmer sans macros ? Lesquels ?

Votre proposition > _____

Y a-t-il des avantages à faire l'édition de liens au moment de l'exécution ? Lesquels ?

Votre proposition > _____

Dans un terminal, regarder le résultat de ldd /bin/ls et objdump -d /bin/ls | less (rapidement pour objdump

In [5]:

Activité : Compilation d'un programme élémentaire

Soit le programme suivant

```
#include <unistd.h>

char *str="Bonjour\n";

int main() {
    write(1,str,8);
    return(0);
}
```

Sauvez-le dans un fichier hello.c puis faites successivement:

```
cpp hello.c -o hello_sans_macros.c
cc -o hello.s -S hello_sans_macros.c
as -o hello.o hello.s
gcc -o hello hello.o # on pourrait utiliser ld directement mais la ligne à taper fait plus de 1000 caractères
```

Comparez ensuite `hello.c` et `hello_sans_macros.c` . Comparez ensuite `hello.s` et le résultat de `objdump -d hello.o` . Trouvez s'il reste des bibliothèques dynamiques à lier au moment de l'exécution avec `ldd ./hello` (3 normalement). Enfin, testez que le programme fonctionne en lançant `./hello` .

Outre la possibilité de faire faire un travail important d'optimisation (et de performance) par l'ordinateur à la compilation, un autre intérêt de la compilation est aussi de pouvoir faire des vérifications statiques sur le programme, avant même de l'exécuter. Par exemple, il est possible de détecter dans certains cas si un morceau de code ne va jamais être exécuté (ce qui est une erreur, dans la plupart des cas). Ce travail d'inspection prend trop de temps pour être fait au lancement du programme (exécution), mais peut très bien s'insérer dans la chaîne de production du logiciel.

Quelques autres langages compilés :

- Le C et beaucoup de ses variantes : C++, C#, Smalltalk
- COBOL
- Fortran
- Parmi des additions plus récentes, on peut au moins compter Go et Rust

Les programmes interprétés

La compilation a des avantages, mais aussi des inconvénients. L'inconvénient le plus flagrant est la *portabilité*, c'est-à-dire la possibilité de faire un programme exécutable sur plusieurs plateformes. Comme le langage machine est spécifique à une famille de processeurs, que les bibliothèques de fonctions pré-programmées le sont aussi... un programme copié sur un système différent ne fonctionnera pas forcément, même si le système d'exploitation reste le même.

Il est possible de trouver la portabilité au niveau source (une recompilation permet d'obtenir un programme fonctionnel sur la nouvelle plateforme), mais plus le nombre de plateformes cibles est grand, plus c'est difficile. Et le travail est à faire sur chaque programme !

Il existe une autre façon de faire un programme portable. Si on fabrique un programme paramétrable, qui lit des instructions dans une sorte de langage universel, et qui agit sur ses zones de mémoire pour simuler des variables, des opérations, etc. en fonction de ce langage universel, il suffit de porter pour une nouvelle plateforme uniquement l'interpréteur de ce langage universel, et tous les programmes fonctionneront. C'est ce qu'on appelle des *interpréteurs*, qui servent ensuite à exécuter des programmes dans des langages *interprétés*. Les fichiers d'instruction dans le langage interprété sont souvent appelés des scripts.

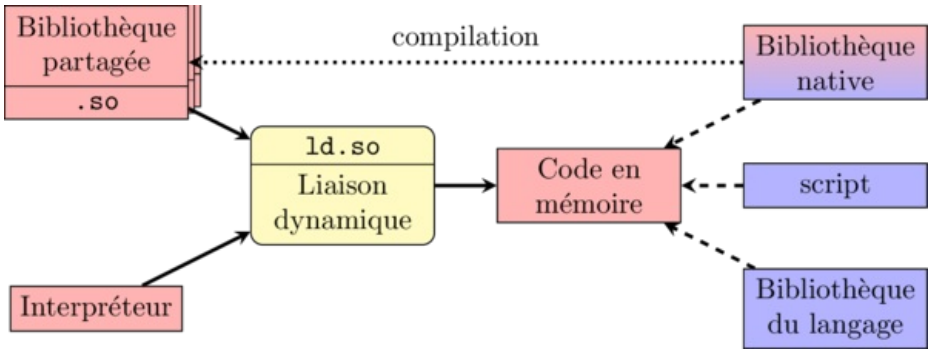
Quelques exemples de tels langages de programmation :

- Le shell (bash par exemple, il en existe d'autres : ksh, zsh, MS-DOS...)
- Python
- Perl, Ruby, PHP, ASP (des langages assez différents, utilisés surtout dans le domaine du web)
- Plusieurs langages de calcul scientifiques : R, Matlab, octave, gnuplot
- Makefile
- SQL dans le domaine des bases de données
- Postscript, pour décrire les instructions graphiques qui pilotent la plupart des imprimantes
- Scheme (une variante de son ancêtre Lisp) dans la catégorie des langages fonctionnels (un type de programmation)

Les programmes interprétés (scripts) sont donc en fait « lus » par l'interpréteur, qui le déchiffre pas à pas et « simule » les effets du programme (mais les effets de la « simulation » sont, eux, biens réels). À aucun moment, c'est le script lui-même qui est lu comme du code ; c'est l'interpréteur qui est exécuté (et son exécution « déroule » le script).

In [6]:

Figure `compilation2*` .



Les programmes compilés à la volée

Il existe une catégorie de langages qui est entre les deux modèles (compilés et interprétés). Au lieu de compiler vers le langage machine propre à chaque processeur, on compile vers une sorte de langage machine universel, qu'on appelle du *bytecode*. Ce langage proche de l'assembleur, très simple, ne tient pas compte d'un certain nombre de limitations de chaque processeur, et surtout ne tient pas compte du codage des instructions.

Ensuite, à l'exécution, le *bytecode* est compilé (au démarrage du programme), la phase d'assemblage étant déjà partiellement faite. Pour obtenir un programme sur une nouvelle plateforme, il suffit donc (en théorie) de créer un compilateur/interpréteur pour le *bytecode* uniquement, sans se préoccuper de toutes les phases qui ont eu lieu avant. L'interpréteur (qui souvent compile à l'exécution des morceaux de codes) est appelé une *machine virtuelle* (ce mot a d'autres sens). Ainsi la *Java Virtual Machine* (JVM) permet de faire tourner le code produit par le langage Java. Les langages à *bytecode* comprennent au moins :

- Java (et sa JVM)
- .NET (et le CLR)

Il est à remarquer que la frontière est de plus en plus mince entre ces types de programme. Par exemple, certains langages interprétés, peuvent aussi être compilés (pour plus d'efficacité) :

- Lisp (variante CLisp)
- Caml (variante Ocaml)
- Python (pour les modules)

Il y a même des langages principalement interprétés, dont seules les fonctions les plus utilisés (on compte au fur et à mesure de l'exécution du programme) sont compilées (au bout d'un moment, certains morceaux de scripts deviennent donc vraiment du code qui est exécuté directement sur le processeur). C'est le cas, en particulier, de *Javascript*. La différence de performance devient donc de moins en moins significative.

In [7]:

Activité : Interprété ou compilé

Récapituler les principaux avantages d'un langage compilé ou interprété

Votre proposition > _____

Vie et mort des processus

Un processus ne vit pas de façon uniforme (lancement, travail, arrêt). Dans un système moderne, le lancement, le travail et l'arrêt sont soumis à des cycles. Nous allons voir deux cycles de façon superficielle : comment on crée et détruit un processus, et ce qui se passe pendant que le processus travaille.

La création d'un processus, sous Unix, ne se fait pas à partir de rien. Un processus est créé par copie d'un autre processus. Plus tard, cette copie continue à travailler indépendamment. Ensuite, lorsque le travail est terminé, elle est détruite.

Le cycle de vie d'un processus peut donc être décrit comme suit:

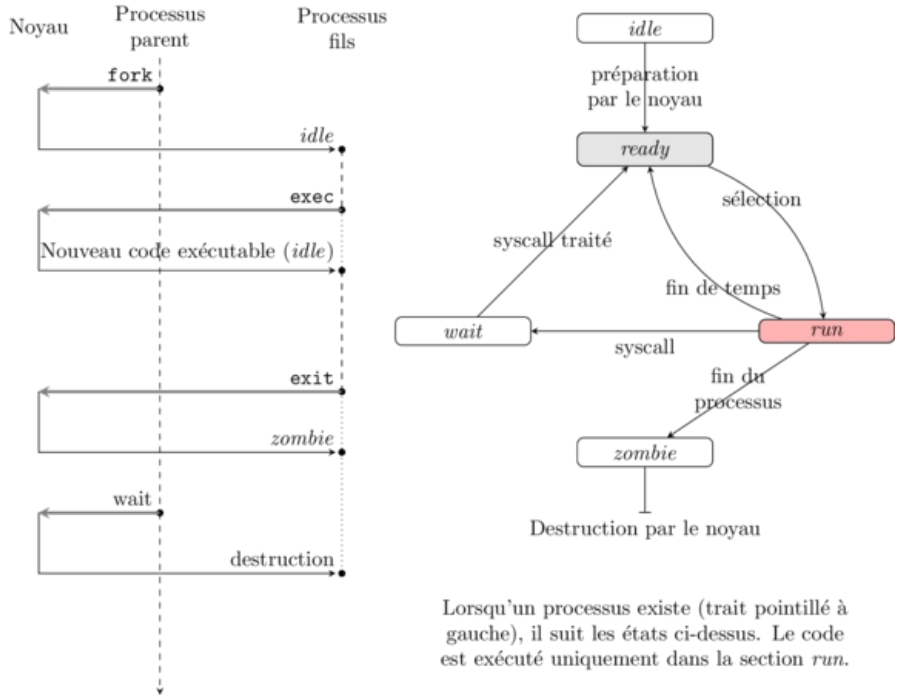
- Le processus (qui va devenir le *processus parent*) fait un appel système `fork` pour se dédoubler. Il y a maintenant deux processus.
- Le processus nouveau (le *processus fils* et le *processus parent*) font leur travail. Pour le moment ils sont sur le même code exécutable.
- Le *processus fils* (souvent) change de code exécutable pour exécuter un autre code. Cela se fait à travers l'appel système `exec` qui permet à un processus de détruire le code exécutable courant et de le faire remplacer par un autre.
- Au bout d'un moment, un processus termine son travail, et déclare volontairement qu'il a terminé par l'appel système `exit` . Il devient alors un *zombie*, et un message est envoyé à son *processus parent*.
- Lorsque le *processus parent* lance l'appel système `wait` (avec le numéro de son fils, ou une valeur qui veut dire « n'importe quel enfant »), il est informé de la valeur de retour du fils. Le système détruit alors le zombie. Le processus a terminé son cycle.

Une fois qu'il est lancé, un processus ne fait pas que fonctionner non plus. À cause du fonctionnement préemptif des systèmes d'exploitation, un processus se voit parfois arrêter, puis repartir. Il est arrêté soit de façon volontaire (par exemple, lorsqu'il fait de lui-même un appel système) ou de façon involontaire (par exemple lorsqu'il arrive à cours de temps de calcul). Il est alors mis dans une file d'attente spéciale, dont il ne sortira que lorsqu'il sera à nouveau prêt à fonctionner (par exemple, lorsque son appel système aura été traité).

En combinant les deux, on obtient le schéma suivant :

In [8]:

Figure `process-life*` .



Lorsqu'un processus existe (trait pointillé à gauche), il suit les états ci-dessus. Le code est exécuté uniquement dans la section *run*.

In [9]:

Activité : Le cycle de vie du processus

Repérer sur le schéma de droite les moments où le processus est dans une file d'attente ordonnée, et les moments où il est juste dans la table des processus à attendre qu'une condition externe lui permette de changer d'état

Votre proposition > _____

Un shell veut lancer un nouveau programme (par exemple `/bin/ls`). Lister dans l'ordre les appels systèmes qui vont être effectués.

Votre proposition > _____

L'arbre des processus

Chaque processus est identifié, de façon unique, par un identifiant appelé PID. Comme chaque processus est engendré par un parent, on peut également donner à chaque processus l'identifiant de son parent, appelé PPID.

À partir de ces deux données, on peut donc construire une structure d'arbre sur les processus.

Les données sur les processus sont accessibles à travers une commande principale `ps` . Plusieurs autres permettent de donner des informations similaires mais plus agréables à lire :

- `top` permet un affichage en temps réel (trié par utilisation du processeur par défaut)
- `pstree` permet d'afficher l'arbre des processus directement

L'usage de la commande `ps` est très simple (elle marche sans arguments), mais un certain nombre d'arguments viennent augmenter son utilité :

In [10]:

```
%%sh
if [ -x /usr/bin/pstree ]; then
    COLUMNS=100
    export COLUMNS
    pstree -U
else
    echo "Désolé, pstree n'est pas disponible sur ce système"
fi
```

```
systemd─┬─AgentAntidote───AgentAntidote.b───22*[{AgentAntidote.b}]
        ├─ModemManager───2*[{ModemManager}]
        ├─NetworkManager─┬─dhclient
        │                └─2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─alsactl
        ├─atd
        ├─avahi-daemon───avahi-daemon
        ├─boltd───2*[{boltd}]
```


├─colord───2*[{colord}]
├─cron
├─cups-browsed───2*[{cups-browsed}]
├─cupsd───{cupsd}
├─dbus-daemon
├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───3*[{Xorg}]
│ │ │ ├─gnome-session-b─┬─AgentConnectix───AgentConnecti+
│ │ │ │ ├─evolution-alarm───5*[{evolutio+
│ │ │ │ ├─gnome-shell─┬─chrome─┬─2*[cat]
│ │ │ │ │ │ ├─chrome─+++
│ │ │ │ │ │ ├─chrome─+++
│ │ │ │ │ │ ├─chrome-+
│ │ │ │ │ │ ├─chrome-+
│ │ │ │ │ │ └─30*[{ch+
│ │ │ │ │ ├─java───35*[{java+
│ │ │ │ │ └─21*[{gnome-shell+
│ │ │ │ ├─gnome-software───11*[{gnome-so+
│ │ │ │ ├─gsd-a11y-settin───3*[{gsd-a11y+
│ │ │ │ ├─gsd-clipboard───2*[{gsd-clipbo+
│ │ │ │ ├─gsd-color───3*[{gsd-color}]
│ │ │ │ ├─gsd-datetime───3*[{gsd-datetim+
│ │ │ │ ├─gsd-disk-utilit───2*[{gsd-disk+
│ │ │ │ ├─gsd-housekeepin───3*[{gsd-hous+
│ │ │ │ ├─gsd-keyboard───3*[{gsd-keyboar+
│ │ │ │ ├─gsd-media-keys───4*[{gsd-media+
│ │ │ │ ├─gsd-mouse───3*[{gsd-mouse}]
│ │ │ │ ├─gsd-power───3*[{gsd-power}]
│ │ │ │ ├─gsd-print-notif───2*[{gsd-prin+
│ │ │ │ ├─gsd-rfkill───2*[{gsd-rfkill}]
│ │ │ │ ├─gsd-screensaver───2*[{gsd-scre+
│ │ │ │ ├─gsd-sharing───3*[{gsd-sharing}+
│ │ │ │ ├─gsd-smartcard───4*[{gsd-smartc+
│ │ │ │ ├─gsd-sound───3*[{gsd-sound}]
│ │ │ │ ├─gsd-wacom───3*[{gsd-wacom}]
│ │ │ │ ├─gsd-xsettings───3*[{gsd-xsetti+
│ │ │ │ ├─tracker-miner-a───4*[{tracker-+
│ │ │ │ ├─tracker-miner-f───4*[{tracker-+
│ │ │ │ ├─zeitgeist-datah───10*[{zeitgei+
│ │ │ │ └─3*[{gnome-session-b}]
│ │ │ └─2*[{gdm-x-session}]
│ │ └─2*[{gdm-session-wor}]
│ └─2*[{gdm3}]
├─gdomap
├─gnome-keyring-d───3*[{gnome-keyring-d}]
├─gsd-printer───2*[{gsd-printer}]
├─irqbalance───{irqbalance}
├─libvirtd───17*[{libvirtd}]
├─master─┬─pickup
│ ├─qmgr
│ └─tlsmgr
├─minissdpd
├─ntpd───{ntpd}
├─polkitd───2*[{polkitd}]
├─qemu-system-x86───5*[{qemu-system-x86}]
├─rsyslogd───3*[{rsyslogd}]
├─rtkit-daemon───2*[{rtkit-daemon}]
├─sshd
├─systemd─┬─(sd-pam)
│ ├─at-spi-bus-laun─┬─dbus-daemon
│ │ └─3*[{at-spi-bus-laun}]
│ ├─at-spi2-registr───2*[{at-spi2-registr}]
│ ├─dbus-daemon
│ ├─dconf-service───2*[{dconf-service}]
│ ├─dleyna-renderer───2*[{dleyna-renderer}]
│ ├─evolution-addre───5*[{evolution-addre}]
│ ├─evolution-calen───9*[{evolution-calen}]
│ ├─evolution-sourc───3*[{evolution-sourc}]
│ ├─gnome-shell-cal───5*[{gnome-shell-cal}]
│ ├─gnome-terminal-─┬─bash───neomutt
│ │ ├─9*[bash]
│ │ ├─bash───gnome-subtitles───5*[{gnome-subtitles}]
│ │ ├─bash───jupyter-noteboo─┬─python3─┬─sh───pstree
│ │ │ │ └─11*[{python3}]
│ │ │ └─2*[{jupyter-noteboo}]
│ │ └─3*[{gnome-terminal-}]
│ ├─goa-daemon───3*[{goa-daemon}]
│ ├─goa-identity-se───3*[{goa-identity-se}]
│ ├─gpg-agent───scdaemon───2*[{scdaemon}]
│ ├─gvfs-afc-volume───3*[{gvfs-afc-volume}]
│ ├─gvfs-goa-volume───2*[{gvfs-goa-volume}]
│ ├─gvfs-gphoto2-vo───2*[{gvfs-gphoto2-vo}]
│ ├─gvfs-mtp-volume───2*[{gvfs-mtp-volume}]
│ ├─gvfs-udisks2-vo───2*[{gvfs-udisks2-vo}]

```
│          ├─gvfsd─┬─gvfsd-burn───2*[{gvfsd-burn}]
│          │       ├─gvfsd-dnssd───2*[{gvfsd-dnssd}]
│          │       ├─gvfsd-http───2*[{gvfsd-http}]
│          │       ├─gvfsd-network───3*[{gvfsd-network}]
│          │       ├─gvfsd-smb-brows───3*[{gvfsd-smb-brows}]
│          │       ├─gvfsd-trash───2*[{gvfsd-trash}]
│          │       └─2*[{gvfsd}]
│          ├─gvfsd-fuse───5*[{gvfsd-fuse}]
│          ├─gvfsd-metadata───2*[{gvfsd-metadata}]
│          ├─nautilus───4*[{nautilus}]
│          ├─pulseaudio───3*[{pulseaudio}]
│          ├─seahorse───3*[{seahorse}]
│          ├─ssh-agent
│          ├─tracker-extract───16*[{tracker-extract}]
│          ├─tracker-store───4*[{tracker-store}]
│          ├─zeitgeist-daemo───2*[{zeitgeist-daemo}]
│          └─zeitgeist-fts───2*[{zeitgeist-fts}]
├─systemd-journal
├─systemd-logind
├─systemd-udevd
├─udisksd───4*[{udisksd}]
├─unattended-upgr───{unattended-upgr}
├─upowerd───2*[{upowerd}]
├─virtlogd
└─wpa_supplicant
```

In [11]:

Activité : Comparer les options de ps

Comparer le résultat de la commande `ps` avec les options suivantes :

- `ps`
- `ps -e`
- `ps -e -f`
- `ps -e -f --forest`
- `ps -o pid:8,ppid:8,user:20,%cpu,args`
- `ps -o pid:8,ppid:8,user:20,%cpu,args --sort=-%cpu,pid`

Les signaux

Si la communication inter-processus est un champ entier de la recherche en informatique, la communication la plus basique entre processus est très simple. Les processus peuvent recevoir un *signal* envoyé par le système, et réagir en fonction du *numéro* de ce signal (une valeur entre 1 et 31). Il n'est pas possible d'envoyer une information plus précise par ce biais. Et la réaction pré-programmée est parfois non-modifiable. C'est un système simple, mais qui permet sans configuration des actions basiques.

L'envoi de signaux peut se faire de deux façons :

- Certaines combinaisons de touche du terminal permettent de dire au shell d'envoyer un signal au processus qui est actuellement en *avant-plan*. Par exemple, Control-C envoie le signal 2 au processus, Control-Z envoie le signal 20.
- La commande `kill` permet d'envoyer le signal de son choix (par défaut le signal TERM)

La commande s'appelle comme ceci parce que la plupart de ces signaux ont une action programmée par défaut pour (à réception), tuer le processus visé (plus ou moins proprement). On peut indiquer les signaux par leur numéro ou par leur nom : `kill -INT 12345` et `kill -2 12345` sont identiques.

Les signaux à connaître :

- HUP : généré par la fin de session (arrête le programme)
- INT : généré par control-C, « interruption » (arrête le programme)
- QUIT : généré par la fin de session (arrête le programme)
- KILL : ne peut pas être masqué (arrête le programme)
- PIPE : généré par l'arrêt d'un programme récepteur d'un pipe pour le programme émetteur (arrête le programme)
- CONT : permet à un programme suspendu de reprendre
- STOP et TSTP : permet de suspendre un programme (le deuxième est généré par control-Z)

La gestion des processus dans le shell

La paire CONT — STOP/TSTP permet en particulier de gérer les processus qui utilisent le terminal quand il y en a plusieurs en même temps.

Lorsqu'on lance une commande suivie d'une esperluette (&), la commande est lancée en tâche de fond. Elle ne peut pas accéder à STDIN. Lorsqu'on suspend une commande (pas en tâche de fond), un numéro de job s'affiche

```
user@host:~$ sleep 30
^Z
[1]+  Stoppé                 sleep 30
```

Le numéro de job peut être utilisé ensuite avec les commandes `bg` et `fg` en mettant un `%` devant. `bg %1` passera le job en tâche de fond, et `fg %1` passera le job en *avant-plan*. On ne peut pas passer un job en avant-plan parce que quand un job est en avant-plan, on ne peut pas saisir de commande dans le shell (le clavier est « connecté » au programme, plus au shell).

La commande `sleep` suivi d'un nombre entier de secondes ne fait rien pendant le nombre de secondes indiqué, puis s'arrête.

In [12]:

```
%%sh
echo "La liste des signaux est :"
/bin/kill -L
```

```
La liste des signaux est :
 1 HUP      2 INT      3 QUIT     4 ILL      5 TRAP     6 ABRT     7 BUS
 8 FPE      9 KILL    10 USR1    11 SEGV    12 USR2    13 PIPE    14 ALRM
15 TERM    16 STKFLT  17 CHLD    18 CONT    19 STOP    20 TSTP    21 TTIN
22 TTOU    23 URG     24 XCPU    25 XFSZ    26 VTALRM  27 PROF    28 WINCH
29 POLL    30 PWR     31 SYS
```

In [13]:

Activité : Tester l'envoi de signaux au clavier

En utilisant la commande `sleep` faire la séquence suivante ou équivalent, et comprendre ce qui se passe à chaque fois:

```
sleep 30
# taper Control-C
sleep 60
# taper Control-Z
fg %1
# taper Control-Z
bg %1
sleep 30 &
sleep 30 &
jobs
sleep 30;jobs
```

In [14]:

Activité : Tester l'envoi de signaux par `kill`

En utilisant deux terminaux, faire la séquence suivante ou équivalent, et comprendre ce qui se passe à chaque fois:

```
sleep 60 & # Terminal 1
ps -e -f -o user,pid,args | grep sleep # Terminal 2 : trouver le PID du sleep 60 du terminal 1
MONPID=... # mettre le bon numéro de processus (Terminal 2)
kill -INT $MONPID # ou kill -2 $MONPID (Terminal 2)
# Observer ce qui se passe dans le terminal 1
# On repasse dans le terminal 1
echo '#!/bin/sh' 1> /tmp/a.sh
echo 'echo "Mon PID est $$"' 1>> /tmp/a.sh # la variable spéciale $$ contient le PID
echo 'trap date INT' 1>> /tmp/a.sh
echo "while true; do sleep 1; echo 'coucou'; done" 1>> /tmp/a.sh
sh /tmp/a.sh # Terminal 1
# Observer ce qui se passe dans le terminal 1
# Essayer de l'arrêter avec Control-C
# Essayer de l'arrêter avec un `kill -2` depuis le terminal 2
# L'arrêter avec un `kill -15` depuis le terminal 2
```

Les utilisateurs et les permissions

Un système, plusieurs utilisateurs

Chaque processus, chaque fichier d'un système est catégorisé comme appartenant à un *utilisateur*. Tous les systèmes ne le sont pas, mais la division d'un système en plusieurs utilisateurs permet d'avoir une meilleure étanchéité entre les composantes d'un système, de gérer plusieurs utilsateurs (humains), de permettre une division plus fines des permissions.

Les utilisateurs sont identifiés par un numéro (surprise), qui peut être converti en un nom (souvent appelé *login*). Ils sont également organisés en groupes (mais un utilisateur peut appartenir à plusieurs groupes).

Les utilisateurs sont — au sens de ce cours — des utilisateurs logiques, et pas des utilisateurs physiques. Il est assez fréquent d'avoir des sous-systèmes d'un serveur qui sont exécutés chacun sous l'identité d'un utilisateur différent (par exemple, le service de mail, le service web, etc.)

L'authentification est la procédure qui consiste à donner suffisamment d'éléments au système pour valider que les éléments ont été donnés par une personne autorisée précise ; c'est à distinguer des informations sur l'identité, qui consiste plus en la lecture d'une base de données

Les utilisateurs dans les systèmes UNIX

Les utilisateurs dans les systèmes UNIX sont identifiés par un numéro (UID) et l'appartenance à un ou plusieurs groupes, dont le groupe principale (les groupes eux-mêmes sont identifiés par des numéros : GID).

Les informations sont stockés dans divers endroits, selon les configurations (fichier `/etc/nsswitch.conf`). Toutefois, dans la configuration la plus simple, c'est la paire de fichiers `/etc/passwd` et `/etc/shadow` qui va contenir les informations d'identification (`/etc/passwd`) et d'authentification (`/etc/shadow`).

Dans le système UNIX, tous les utilisateurs sont *a priori* équivalents, sauf un : l'utilisateur de numéro 0, appelé `root`. Cet utilisateur possède, au niveau du système, des droits particuliers (notamment, il peut ignorer les restrictions de permission). Il est reservé à l'administration du système (pour faire les mises à jour, par exemple).

Il est possible de lister toutes les informations en utilisant la commande `getent` (qui permet de trouver toutes les identités). Les données d'identification sont assez sommaires : *login*, *mot de passe* (dans la plupart des systèmes, il doit être caché et n'est accessible que dans la table `shadow`, qui n'est lisible que par `root`), *UID*, *GID*, *répertoire personnel*, *shell*. D'autres informations peuvent exister dans d'autres annuaires (comme un annuaire LDAP), mais elles ne relèvent pas du système d'utilisateurs.

In [15]:

```
%%sh
echo -n "Nombre d'utilisateurs accessibles avec getent : "
getent passwd|wc -l
echo -n "Nombre de groupes accessibles avec getent : "
getent group|wc -l
echo "Informations sur l'utilisateur root :"
getent passwd | grep ^root:

return 0
```

```
Nombre d'utilisateurs accessibles avec getent : 46
Nombre de groupes accessibles avec getent : 78
Informations sur l'utilisateur root :
root:x:0:0:root:/root:/bin/bash
```

In [16]:

```
%%sh
echo "Votre groupe principal est \"$(id -gn)\" --- de numéro $(id -g)"
echo "Votre identifiant est \"$(id -un)\" --- de numéro $(id -u)"
echo "Vos groupes sont \"$(id -Gn)\" --- de numéro $(id -G)"
```

```
Votre groupe principal est "users" --- de numéro 100
Votre identifiant est "jcdubacq" --- de numéro 1000
Vos groupes sont "users cdrom floppy audio dip video plugdev netdev libvirt kvm" --- de numéro 100 24 25 29 30 44
46 108 122 123
```

Le système d'utilisateurs a de l'influence sur le système à travers deux mécanismes :

- Chaque processus tourne avec l'identité d'un utilisateur (et d'un seul)
- Chaque fichier ou répertoire appartient à un utilisateur et a des permissions qui concernent chaque classe d'utilisateurs du système. Ces permissions peuvent empêcher l'accès au contenu du fichier ou du répertoire.

Ainsi, si un fichier est interdit en lecture à toute autre personne que son propriétaire, aucun processus qui n'appartient pas au propriétaire ne pourra en lire les données (ou même l'ouvrir).

Il est important de connaître le système de permissions.

Les permissions sous UNIX

Les permissions sous UNIX sont de trois sortes, selon le type d'élément (répertoire ou fichier). Pour les fichiers, on a le droit de :

- **lire** le contenu du fichier
- **modifier** le contenu du fichier (et en particulier le remplir quand il est vide)
- **exécuter** le fichier (ce qui n'a de sens que si c'est un *script* interprétable ou du *code* exécutable).

Pour les répertoires, les permissions sont les suivantes :

- **consulter le catalogue** du répertoire (on dit souvent **lister**)
- **modifier le catalogue** du répertoire, ce qui est le droit de créer ou supprimer des fichiers dedans
- **traverser le répertoire** ce qui permet d'obtenir l'information sur où est situé un élément du catalogue.

Il est important de se rappeler ce qu'est un chemin (relatif ou absolu) pour bien comprendre la dernière permission. En effet, pour accéder à un fichier, on passe par son *chemin*, qui est composé d'un enchaînement de répertoires (s'il n'y a que le nom du fichier, on passe en fait par le répertoire appelé `.`). Pour accéder à un fichier, il faut non seulement avoir le droit sur le fichier, mais aussi le droit de traverser tous les répertoires qui le précèdent !

Nom symbolique	Poids	Fichier	Répertoire
r	4	Lecture	Consultation
w	2	Modification	Modification
x	1	Exécution	Traversée

Les fichiers exécutables, lorsqu'ils sont le premier argument d'une ligne de commande, sont des programmes. La permission d'exécution doit être accordée pour que ça marche, ainsi que la permission de lecture. Si les premiers octets d'un fichier exécutable sont `#!` suivi d'un chemin (ou `#!`) alors c'est un script (et le chemin doit être celui de l'interpréteur), sinon le programme doit être dans un format reconnu par le noyau.

Les groupements

Il y a trois catégories de permissions pour un même fichier. Ils sont déterminés automatiquement en fonction du *propriétaire* du fichier et du *groupe* propriétaire du fichier.

Pour savoir si une identité (utilisateur, groupe) a accès à un fichier, on regarde si :

- L'utilisateur est le même que le propriétaire. Si oui, alors on utilise la catégorie **propriétaire**.
- Les groupes de l'utilisateur contiennent le groupe propriétaire. Si oui, alors on utilise la catégorie **groupe**.
- Sinon, on utilise la catégorie **autre**.

Cette classification rudimentaire peut être complété par un système plus complexe (ACL) qui est en dehors de ce cours, qui permet des permissions individuelles.

Pour connaître l'identité du propriétaire d'un processus ou d'un fichier

- Les commandes top et ps affichent le nom du propriétaire des processus.
- La commande `ls` avec l'option `-l` affiche le nom et le groupe du propriétaire d'un fichier ou d'un répertoire, de même que la commande `stat` .
- Les UID et GID sont enregistrés dans le fichier d'administration /etc/passwd ou d'autres mécanismes

Ainsi, chaque fichier a trois paquets de trois permissions, que l'on peut décoder. Par exemple, une permission `rw-r-----` sur un fichier signifie que le propriétaire peut lire et modifier le fichier (mais pas l'exécuter), un utilisateur du même groupe peut le lire (mais pas le modifier ni l'exécuter) et sinon, rien n'est possible.

Les permissions sont regroupées par paquets de 3, et sont parfois codées numériquement selon le code donné plus haut : `rw-r-----` est ainsi équivalent numériquement à `640`.

Changer les permissions

Les permissions se changent avec la commande `chmod`.

Il y a deux usages de cette commande. L'usage *symbolique* consiste à exprimer sous formes de lettres les permissions que l'on donne ou retire. Par exemple, `chmod ug-wx fichier1 fichier2 fichier3` enlève les droits d'écriture et d'exécution à l'utilisateur et à son groupe sur les trois fichiers. Et `chmod o+r fichier` donne les droits de lecture à tous les utilisateurs *autres* sur le fichier.

Les lettres à utiliser avant le signe `-` ou `+` sont à piocher dans `ugoa` : utilisateur, groupe, autres, tous (un raccourci pour `ugo`).

Le deuxième usage (dit *numérique*) consiste tout simplement à dire quels sont les modes à mettre sous forme numérique. Par exemple si on veut mettre un fichier en lecture seule pour tout le monde (utilisateur, groupe et autres), on utilisera `chmod 444 fichier`.

In [28]:

Activité : Les bonnes permissions selon les activités

En utilisant les deux notations, dites quelle est la permission la plus adaptée pour le cas proposé.

Un fichier public, à modification uniquement par le propriétaire

Votre proposition > _____

Solution: Notation numérique : 640, notation lettres : `rw-r-----`

Un fichier privé

Votre proposition > _____

Solution: Notation numérique : 600, notation lettres : `rw-------`

Un fichier de travail en groupe

Votre proposition > _____

Solution: Notation numérique : 660, notation lettres : `rw-rw----`

Un script utilisable par tout les système

Votre proposition > _____

Solution: Notation numérique : 755, notation lettres : `rwxr-xr-x`

Un script personnel avec un mot de passe écrit en clair dedans

Votre proposition > _____

Solution: Notation numérique : 700, notation lettres : `rwx------`

Un répertoire à accès en lecture publique

Votre proposition > _____

Solution: Notation numérique : 755, notation lettres : `rwxr-xr-x`

Un répertoire de travail en groupe

Votre proposition > _____

Solution: Notation numérique : 770, notation lettres : `rwxrwx---`

Un répertoire privé

Votre proposition > _____

Solution: Notation numérique : 700, notation lettres : `rwx------`

Un répertoire que seuls ceux qui connaissent le nom des fichiers peuvent exploiter (et le propriétaire)

Votre proposition > _____

Solution: Notation numérique : 711, notation lettres : `rwx--x--x`

In [44]:

Activité : Vérifier les permissions

Au moyen de la commande id, affichez votre UID et votre GID ? Comparez-le avec celui de votre voisin de table. Qu'en concluez-vous ? Comparez-les avec celui de l'utilisateur root. Qu'en concluez-vous ?

Votre proposition > _____

Solution: Les identifiants des étudiants et leur groupes sont construits mécaniquement d'après l'annuaire, et ne sont que des nombres. Ce n'est pas le cas pour les enseignants ou pour les utilisateurs systèmes.

Quels sont vos droits sur le répertoire racine `/` , `/root` , `/tmp` , sur votre répertoire `~/` , et celui de votre voisin de table `~/../login_voisin` ?

Votre proposition > _____

Solution: Normalement, pas de droit d'écriture mais droit de lecture sur `/` . Les groupes des élèves étant différents, ils sont traités comme des autres et pas comme un groupe et donc ne peuvent pas voir par défaut le contenu des répertoires des voisins. Pour `/tmp` , le réglage est un peu plus compliqué à cause du bit t qui est une propriété particulière dont on ne parle pas dans ce cours, qui empêche certaines suppressions de fichier. Attention à bien faire `ls -ld` ou `stat` pour voir les droits.

Pouvez-vous lire les données contenue dans le répertoire de votre voisin. Quelle commande permettrait de le faire ? Qui doit lancer la commande ?

Votre proposition > _____

Solution: Il faut que le voisin l'autorise par un `chmod a+x ~` ou `chmod a+rx ~` (les fichiers sont par défaut lisibles de tous, normalement, mais le répertoire personnel n'est pas traversable.

Donnez les commandes octale et alphanumérique de changement de droits permettant :

- de créer et d'autoriser aux membres de votre groupe et aux "autres" l'accès en lecture aux images du répertoire `~/public_html/tempo`
- de créer et donner les droits d'écriture aux membres de votre groupe uniquement (et vous) sur le fichier `~/public_html/tempo/bash.txt`
- de vous (le propriétaire) retirer toute possibilité de supprimer directement le fichier ci-dessus.

Votre proposition > _____

Solution:

- `mkdir -p ~/public_html/tempo ; chmod a+x ~ ~/public_html ; chmod a+rx ~/public_html/tempo`
- `touch ~/public_html/tempo/bash.txt ; chmod a-x,g+rw,o-rw ~/public_html/tempo/bash.txt`
- `chmod -w ~/public_html/tempo` (c'est le seul moyen, les personnes ayant droit d'écriture dans le répertoire peuvent effacer)

Imaginez comment donner à votre voisin un accès sous votre répertoire personnel à un répertoire dans lequel il aurait les droits d'écriture sur un fichier spécifique, que vous ne pourriez vous que lire (mais pas modifier). Il ne doit pas pouvoir créer un autre fichier chez vous. Comment faites vous pour effacer ce fichier ?

Votre proposition > _____

Solution: Il faut juste créer un fichier, donner les droits d'écriture dessus, et s'enlever les droits d'écriture. Il ne faut pas donner les droits d'écriture sur le répertoire. D'autres solutions sont imaginables.