Linux -- Cours 3 -- Bases des systèmes d'exploitation

Halim Djerroud

LISV
Laboratoire d'ingénierie
des systèmes de Versailles

UVSQ
université PARIS-SACLAY

révision : 1.2

Plan de la séance

1. Les processus.
2. Variables d'environnement : `PATH`, `HOME`, `USER`.
3. Scripts Bash : création, exécution, droits d'exécution.
4. Structures conditionnelles et de contrôle : `if`, `for`, `while`.
5. Commandes réseau : `ping`, `traceroute`, `wget`, `curl`, `netstat`, `nmap`.

Processus sous Unix/Linux

Gestion des processus

Objectifs du chapitre :

- Comprendre ce qu'est un processus et son cycle de vie.
- Identifier les différents types de processus.
- Visualiser et surveiller les processus actifs.
- Interagir avec eux : avant-plan / arrière-plan, priorités, signaux.
- Relier la gestion des processus à l'administration système.

Qu'est-ce qu'un processus ?

- **Programme** : suite d'instructions stockée sur disque.
- **Processus** : instance d'un programme en cours d'exécution (en mémoire).
- Chaque processus est identifié par un **PID** (Process ID).
- Deux types principaux :
  - **Processus système** : créés au lancement de l'OS ou exécutés en tâche de fond (*daemons*).
  - **Processus utilisateur** : lancés depuis un terminal ou un script.
- Attributs importants :
  - UID (utilisateur propriétaire)
  - Priorité / état (`running`, `sleeping`, `zombie`...)
  - Ressources utilisées (mémoire, CPU, fichiers ouverts)
- Processus père / fils : hiérarchie (ex. `init` ou `systemd` = ancêtre de tous).

Attributs d'un processus

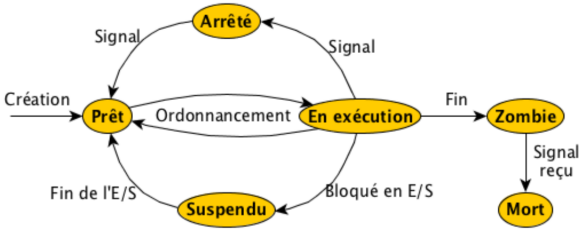
- PID : identifiant unique.
- PPID : identifiant du processus parent.
- UID / EUID : propriétaire et droits effectifs.
- GID / EGID : groupe propriétaire.
- TTY : terminal associé.
- Autres : état, priorité (NI), consommation mémoire/CPU, répertoire de travail. . .

Cycle de vie d'un processus

- **R** : Running (en cours d'exécution).
- **S** : Sleeping (en attente).
- **T** : Stopped (suspendu).
- **Z** : Zombie (terminé mais non libéré par le père).
- **D** : Uninterruptible sleep (bloqué E/S).

À retenir

Un processus peut être créé, suspendu, réveillé et détruit. Un zombie = processus mort mais encore présent dans la table.



Lister et surveiller les processus

Commandes principales :

```
$ ps              # processus du terminal courant
$ ps -a           # processus de la console courante
$ ps aux          # tous les processus (BSD style)
$ ps -ef          # tous les processus (format détaillé SYSV)
$ ps -ejH         # affichage en arbre
$ top             # surveillance en temps réel (CPU/mémoire)
$ htop            # version interactive (si installée)
$ atop            # version avancée (si installée)
```

Exemple : rechercher un processus spécifique

```
$ ps aux | grep firefox
user   1234  5.0  2.1  ...  /usr/lib/firefox/firefox
```

Hiérarchie et recherche de processus

Visualiser la hiérarchie : `pstree`

```
$ pstree
systemd---NetworkManager---dhclient
        |--sshd---bash---emacs
        |--gnome-shell
```

- Affiche les processus sous forme d'arbre.
- Permet de voir les relations père/fils.
- Utile pour repérer quels programmes dépendent d'un autre.

Rechercher un processus : `pgrep`

```
$ pgrep bash
25510

$ pgrep -a Xorg
1332 /usr/bin/X -core :0 -seat seat0 ...
```

- Recherche un processus par motif.
- `-a` : affiche aussi la ligne de commande.
- Plus pratique que `ps | grep`.

Astuce

Combinez `pstree` pour la vue d'ensemble et `pgrep` pour la recherche ciblée.

Avant-plan et arrière-plan

Lancer un processus :

```
$ firefox &          # lance en arrière-plan
$ sleep 100000       # lance au premier plan
^Z                   # suspend avec Ctrl+Z
[1]+ Arrêté sleep 100000
```

Gérer les jobs :

```
$ jobs               # liste les jobs en arrière-plan/suspendus
$ bg %1              # reprend le job n°1 en arrière-plan
$ fg %1              # ramène le job n°1 au premier plan
```

Astuce

`&` → lance directement en arrière-plan.

`Ctrl+Z` → suspend un processus au premier plan.

`fg`/`bg` → permettent de reprendre ou déplacer un job.

Priorité et ordonnancement

- La colonne `NI` dans `top/htop` indique la **nice value**.
- Plus la valeur est basse, plus le processus est prioritaire.
- Valeurs possibles : de -20 (très prioritaire) à 19 (moins prioritaire).

```
# Lancer un programme avec une priorité réduite
$ nice -n 10 firefox

# Modifier la priorité d'un processus existant
$ renice -n 5 -p 1234
```

Tout est fichier : le répertoire /proc

Par processus (`/proc/<PID>`) :

- `cmdline` : commande de lancement.
- `status` : état, UID, mémoire, signaux.
- `exe` : lien vers l'exécutable.
- `fd/` : descripteurs de fichiers ouverts.

Exemple : explorer un processus

```
$ ps -e | grep bash
1234 ?   00:00:00 bash

$ ls /proc/1234
cmdline  cwd  environ  exe
fd/      maps  status  ...
```

Global système :

- `/proc/uptime` : temps depuis le démarrage.
- `/proc/meminfo` : informations mémoire.
- `/proc/cpuinfo` : infos processeur.
- `/proc/loadavg` : charge système.

Astuce

Sous Linux, **tout est fichier** : même les processus et les ressources système peuvent être explorés via `/proc`.

Signaux Unix et terminaison de processus

Envoyer des signaux :

```
$ kill 1234          # SIGTERM (arrêt propre par défaut)
$ kill -9 1234       # SIGKILL (forcé, non interceptable)
$ killall firefox    # termine tous les processus "firefox"
$ pkill firefox      # idem avec recherche par nom
$ pkill -u hdd    # termine tous les processus d'un utilisateur
```

Signaux courants :

- `SIGTERM (15)` : demande d'arrêt propre (par défaut avec kill).
- `SIGKILL (9)` : arrêt immédiat, non interceptable.
- `SIGINT (2)` : interruption clavier (`Ctrl+C`).
- `SIGTSTP (20)` : suspension (`Ctrl+Z`).
- `SIGCONT (18)` : reprise d'un processus suspendu.
- `STOP` : suspension forcée (non interceptable).

Astuce

Privilégier `SIGTERM` avant `SIGKILL`, pour laisser au processus le temps de se terminer proprement.

Liste des signaux

```
$ kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE    14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD    18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO      30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9 56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Signaux Unix à connaître

Signal	Numéro	Description
SIGHUP	1	Déconnexion détectée du terminal associé ou fin du processus parent.
SIGINT	2	Interruption par l'utilisateur (`Ctrl + C`).
SIGQUIT	3	Quitter : signal envoyé par l'utilisateur (`Ctrl + \`).
SIGFPE	8	Erreur mathématique (ex. division par zéro).
SIGKILL	9	Si un processus reçoit ce signal il doit s'arrêter immédiatement. Il ne peut faire aucune opéra- tion de nettoyage/sauvegarde ni éviter ce signal.
SIGALRM	14	Signal envoyé par une horloge (déclenchement d'événement).
SIGTERM	15	Arrêt logiciel (envoyé par défaut par `kill`).

Variables d'environnement

Introduction aux variables d'environnement

Objectifs du chapitre :

- Variables d'environnement et variables Shell.
- Portée et transmission.
- Commandes de gestion : `printenv`, `env`, `set`.
- Création, export et suppression de variables.
- Persistance (`.bashrc`, `/etc/profile`).
- Commandes avancées : `set`, `source`.

Qu'est-ce qu'une variable ?

- Une **variable** est une zone mémoire qui contient une valeur.
- Dans un shell, une variable est une **association nom → valeur**.
- Elle sert à stocker des informations réutilisables par l'utilisateur ou le système.

Exemple simple :

```
$ NAME="Alice"
$ AGE=22
$ echo $NAME
Alice
$ echo $AGE
22
```

Attention

Pas d'espace autour du signe =.

Différence : variable Shell vs variable d'environnement

Variable Shell

- Définie dans le shell courant.
- Visible uniquement dans ce shell.
- Non transmise aux processus enfants.
- Exemple :

```
$ VAR="bonjour"
$ echo $VAR
bonjour
$ sh
$ echo $VAR    # vide
```

Variable d'environnement

- Exportée avec `export`.
- Transmise aux processus enfants.
- Utilisée par le système et les programmes.
- Exemple :

```
$ export VAR="bonjour"
$ sh
$ echo $VAR
bonjour
```

Variables d'environnement vs Variables Shell

Variables d'environnement

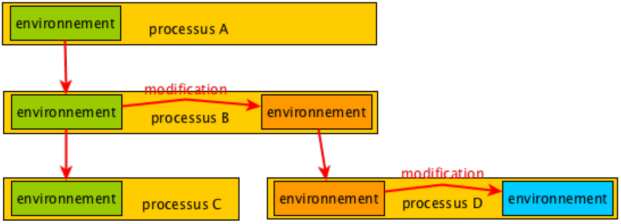
- Définies par le système ou l'utilisateur.
- Transmises automatiquement aux processus enfants.
- Utilisées par le système et les applications.
- Exemples :
  - `$PATH` : chemins de recherche.
  - `$HOME` : répertoire utilisateur.
  - `$USER`, `$SHELL`.

Variables Shell

- Spécifiques au shell courant.
- Définies par l'utilisateur ou dans `/.bashrc`.
- Non transmises aux processus enfants.
- Exemple :
  - `$PS1` : invite de commande.

Portée et transmission

- Chaque processus hérite d'une **copie** de l'environnement de son parent.
- Les modifications d'un enfant ne remontent pas au parent.
- Seules les variables exportées (`export`) sont transmises.



Afficher l'environnement

- L'**environnement** d'un shell est l'ensemble des variables disponibles pour ce shell et ses processus enfants.
- Commandes utiles :

```
$ printenv            # affiche tout l'environnement
$ printenv PATH       # affiche une seule variable
$ echo $PATH          # autre manière d'afficher
$ printenv | grep USER    # chercher une variable
```

Astuce

`printenv` est pratique pour voir l'environnement complet, tandis que `echo $VAR` permet de vérifier une seule variable.

Créer et exporter des variables

- Une **variable Shell** est définie avec `VAR=valeur`.
- Pour qu'elle soit visible par les processus enfants, il faut l'**exporter**.

```
# Créer une variable Shell
$ VAR="ma valeur"
$ echo $VAR
ma valeur

# Exporter dans l'environnement
$ export VAR
$ printenv VAR
ma valeur

# Supprimer une variable
$ unset VAR
```

Différence

Sans `export`, la variable n'existe que dans le shell courant. Avec `export`, elle est héritée par les processus enfants.

Modifier la variable PATH

- `PATH` contient la liste des répertoires où le shell cherche les programmes exécutables.
- Modifier `PATH` permet d'ajouter de nouveaux répertoires.

```
$ echo $PATH
/home/user/bin:/usr/bin:/bin

# Ajouter le répertoire courant (.)
$ PATH=".:$PATH"
$ export PATH
```

Attention

Mettre `.` (répertoire courant) en tête de PATH est risqué : cela peut exécuter par erreur un script malveillant présent dans le dossier.

Commandes avancées

- `set` :
  - Affiche toutes les variables Shell et fonctions.
  - Peut modifier le comportement du shell.
  - Exemples :
    - `set -o noclobber` : empêche d'écraser un fichier existant avec >.
    - `set -o notify` : avertit quand une tâche en arrière-plan se termine.

- `source` (ou `.`) :
  - Exécute un script dans le shell courant.
  - Utile pour charger une configuration sans lancer un nouveau processus.
  - Exemple :

```
$ source ~/.bashrc
```

Persistance des variables

- Par défaut, une variable existe seulement **tant que le shell est ouvert**.
- Pour la rendre **permanente**, il faut l'ajouter dans un fichier de configuration relu à chaque ouverture de session.
- Fichiers courants :
  - `~/.bashrc` : configuration spécifique à l'utilisateur.
  - `/etc/profile` : configuration globale du système.
  - `~/.profile`, `~/.bash_profile` : parfois utilisés selon les distributions.

Exemple :

```
# Dans ~/.bashrc
export PATH=$PATH:/home/user/bin

# Recharger immédiatement le fichier sans redémarrer
$ source ~/.bashrc
```

Astuce

Placer une variable dans `~/.bashrc` la rend disponible à chaque nouvelle session de l'utilisateur.

Quelques commandes et variables à retenir

- printenv
- env
- set
- unset
- export
- declare -x
- readonly
- alias

- echo $VAR
- echo $PATH
- echo $HOME
- echo $USER
- echo $SHELL
- echo $PWD
- echo $LANG
- echo $?

- PATH
- HOME
- USER
- SHELL
- PWD
- PS1
- EDITOR
- LANG

- source ~/.bashrc
- . ~/.bashrc
- source /etc/profile
- VAR=valeur
- VAR="texte"
- export VAR
- unset VAR
- printenv | grep VAR

Scripts Bash

Introduction aux scripts Bash

Objectifs du chapitre :

- Comprendre ce qu'est un script Bash.
- Créer un script simple (fichier texte + shebang).
- Exécuter un script de différentes manières.
- Gérer les droits d'exécution (`chmod`).
- Distinguer `bash`, `./script`, `source`.
- Automatiser des tâches simples.

Qu'est-ce qu'un script Bash ?

- Un **script Bash** est un fichier texte contenant une suite de commandes shell.
- Il permet d'automatiser des tâches répétitives.
- Première ligne : le **shebang**, qui précise l'interpréteur.

```
#!/bin/bash
# Mon premier script
echo "Bonjour, ce script fonctionne !"
```

Créer et exécuter un script

1. Créer un fichier texte avec `nano` ou `vim`.
2. Ajouter le **shebang** en première ligne.
3. Sauvegarder et donner les droits d'exécution.

Exemple :

```
$ nano monscript.sh
#!/bin/bash
echo "Hello World"

# Donner le droit d'exécution
$ chmod +x monscript.sh

# Lancer le script
$ ./monscript.sh
Hello World
```

Exécution d'un script : différentes manières

- En appelant directement le fichier (si exécutable) :

```
$ ./monscript.sh
```

- Avec l'interpréteur `bash` :

```
$ bash monscript.sh
```

- Avec `source` (dans le shell courant) :

```
$ source monscript.sh
```

Différence

`bash script.sh` → lance un sous-processus. `source script.sh` → exécute dans le shell courant.

Droits d'exécution

- Un script doit avoir le droit **x** (exécution).
- Vérifier avec :

```
$ ls -l monscript.sh
-rw-r--r-- 1 user user 45 sep 14 10:00 monscript.sh
```

- Ajouter le droit d'exécution :

```
$ chmod +x monscript.sh
$ ls -l monscript.sh
-rwxr-xr-x 1 user user 45 sep 14 10:00 monscript.sh
```

Structure d'un script Bash

- Un script est un fichier texte qui contient :
  1. **Shebang** : indique quel interpréteur utiliser.
  2. **Commandes** : mêmes commandes que dans le terminal.
  3. **Commentaires** : lignes commençant par #.

Exemple :

```
#!/bin/bash
# Script de démonstration
echo "Utilisateur : $USER"
echo "Répertoire courant : $PWD"
date
```

Variables et arguments dans un script

- On peut utiliser des variables comme dans le shell :

```
NOM="Alice"
echo "Bonjour $NOM"
```

- Les **arguments** sont accessibles avec :
  - `$1`, `$2` ... : premier, deuxième argument.
  - `$@` : tous les arguments.
  - `$#` : nombre d'arguments.
  - `$$` : Numéro du processus courant
  - `$!` Numéro du dernier processus lancé en arrière-plan
  - `$?` :Statut (état final) de la dernière commande

Exemple :

```
#!/bin/bash
echo "Nom du script : $0"
echo "Premier argument : $1"
echo "Nombre d'arguments : $#"
```

Exemple pratique : script d'informations

```
#!/bin/bash
# infos.sh : affiche quelques infos système

echo "Utilisateur : $USER"
echo "Répertoire : $PWD"
echo "Date : $(date)"
echo "Machine : $(hostname)"
```

Exécution :

```
$ chmod +x infos.sh
$ ./infos.sh
Utilisateur : alice
Répertoire : /home/alice
Date : Sat Sep 14 10:30:00 CEST 2025
Machine : pc-linux
```

Bonnes pratiques

- Toujours mettre le **shebang** : `#!/bin/bash`.
- Ajouter des **commentaires** pour expliquer le code.
- Vérifier les **droits d'exécution** (`chmod +x`).
- Retourner un **code de sortie** avec `exit 0` (succès) ou `exit 1` (erreur).
- Tester les scripts étape par étape.

Structures conditionnelles et de contrôle

Introduction aux structures de contrôle en Bash

Objectifs du chapitre :

- Comprendre les structures conditionnelles (`if`, `else`, `elif`).
- Savoir utiliser les boucles (`for`, `while`, `until`).
- Manipuler les tests avec `[ ]` et `test`.
- Écrire des scripts plus dynamiques et interactifs.

La structure `if`

- Permet d'exécuter des commandes si une condition est vraie.
- Syntaxe de base :

```
if [ condition ]
then
    commandes
fi
```

Exemple :

```
if [ $USER = "root" ]
then
    echo "Vous êtes administrateur"
else
    echo "Vous êtes un utilisateur normal"
fi
```

La commande `test`

- La commande `test` évalue une condition et retourne :
  - `0` si la condition est vraie (succès),
  - `1` si la condition est fausse (échec).
- Équivalent à utiliser les crochets `[ ]`.

```
# Avec test
test $USER = "root"
# Avec crochets (équivalent)
[ $USER = "root" ]
# Exemple avec if
if test -f /etc/passwd
then
    echo "Le fichier existe"
fi
```

Astuce

Toujours mettre des espaces autour de la condition :

```
[ $a -eq 5 ] # CORRECT
[$a -eq 5] # FAUX : ne pas oublier les espaces
```

Tests courants avec `if`

- Comparaison de nombres :
  - `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`
- Comparaison de chaînes :
  - `=`, `!=`, `-z` (vide), `-n` (non vide)
- Tests sur fichiers :
  - `-e` existe, `-f` fichier, `-d` répertoire
  - `-r` lisible, `-w` inscriptible, `-x` exécutable

```
if [ -f /etc/passwd ]
then
    echo "Le fichier existe"
fi
```

La structure `elif`

Principe :

- Permet de tester plusieurs conditions successives.
- S'utilise quand un simple `if/else` ne suffit pas.

Syntaxe générale :

```
if [ condition1 ]
then
    commandes1
elif [ condition2 ]
then
    commandes2
else
    commandes3
fi
```

Exemple pratique :

```
note=15
if [ $note -ge 16 ]
then
    echo "Très bien"
elif [ $note -ge 10 ]
then
    echo "Passable"
else
    echo "Échec"
fi
```

La structure `case`

- Alternative à plusieurs `if/elif`.
- Pratique pour tester la valeur d'une variable.

```
case $variable in
    motif1) commandes ;;
    motif2) commandes ;;
    *) commandes par défaut ;;
esac
```

Exemple : menu simple

```
echo "Choisissez une option :"
read choix
case $choix in
    1) echo "Option 1 sélectionnée" ;;
    2) echo "Option 2 sélectionnée" ;;
    *) echo "Choix invalide" ;;
esac
```

La boucle `for`

- Sert à répéter des commandes pour une liste de valeurs.

```
for var in val1 val2 val3
do
    echo "Valeur : $var"
done
```

Exemple :

```
for fichier in *.txt
do
    echo "Fichier : $fichier"
done
```

La boucle `while`

- Répète des commandes tant qu'une condition est vraie.

Exemple :

```
while [ condition ]
do
    commandes
done
```

Exemple :

```
i=1
while [ $i -le 5 ]
do
    echo "Compteur : $i"
    i=$((i+1))
done
```

La boucle `until`

- Exécute les commandes **jusqu'à** ce que la condition devienne vraie.

Exemple :

```
until [ condition ]
do
    commandes
done
```

Exemple :

```
i=1
until [ $i -gt 5 ]
do
    echo "Compteur : $i"
    i=$((i+1))
done
```

Contrôle des boucles : `break` et `continue`

- `break` : sort immédiatement de la boucle.
- `continue` : passe à l'itération suivante.

```
for i in 1 2 3 4 5
do
    if [ $i -eq 3 ]
    then
        continue    # saute le 3
    fi
    if [ $i -eq 5 ]
    then
        break       # arrête la boucle
    fi
    echo "i = $i"
done
```

Résultat : `i = 1, i = 2, i = 4`

La commande `select` (menu interactif)

- Permet de créer un menu facilement.
- Syntaxe :

```
select var in liste
do
    commandes
done
```

Exemple :

```
select fruit in pomme banane orange
do
    echo "Vous avez choisi : $fruit"
    break
done
```

Bonnes pratiques pour utiliser les structures de contrôle

- **if / else / elif** permettent d'exécuter des actions conditionnelles.
- **for** parcourt une liste de valeurs.
- **while** répète tant qu'une condition est vraie.
- **until** répète jusqu'à ce qu'une condition soit vraie.
- Les tests (`[ ]`) sont essentiels pour les conditions.

Commandes réseau

Introduction aux commandes réseau sous Linux

Objectifs du chapitre :

- Vérifier la connectivité réseau (`ping`, `traceroute`).
- Télécharger des fichiers depuis Internet (`wget`, `curl`).
- Surveiller l'état du réseau (`netstat`).
- Découvrir les services disponibles sur une machine (`nmap`).

ping

- Teste la **connectivité** entre votre machine et une destination.
- Envoie des paquets ICMP pour mesurer :
  - si la machine répond,
  - le **temps aller-retour** (latence),
  - le taux de perte de paquets.

```
# Vérifier la connectivité avec google.com
$ ping google.com

# Envoyer seulement 4 paquets
$ ping -c 4 8.8.8.8
```

À retenir

Si `ping` échoue, le problème peut venir : (1) de la connexion locale, (2) du réseau intermédiaire, ou (3) de la machine distante.

traceroute

- Permet de visualiser le **chemin parcouru** par les paquets.
- Liste tous les routeurs intermédiaires entre vous et la destination.
- Utile pour savoir où se situe un ralentissement ou une coupure.

```
# Afficher les routeurs entre votre PC et fsf.org
$ traceroute fsf.org
```

Astuce

Chaque ligne correspond à un "saut" réseau (*hop*) avec son temps de réponse. Un $\star$ signifie que le routeur n'a pas répondu.

wget

- Télécharge automatiquement des fichiers depuis Internet.
- Supporte les protocoles : HTTP, HTTPS, FTP.
- Peut reprendre un téléchargement interrompu (`-c`).

```
# Télécharger une page web
$ wget http://example.com/index.html

# Télécharger un fichier et le renommer
$ wget -O page.html http://example.com
```

Utilisation typique

Parfait pour récupérer des fichiers dans un script ou pour du téléchargement en masse.

curl

- Outil puissant pour interagir avec des serveurs web.
- Peut :
  - afficher une page web dans le terminal,
  - télécharger un fichier,
  - envoyer des requêtes HTTP (GET, POST. . .).

```
# Afficher le contenu d'une page web
$ curl http://example.com

# Télécharger un fichier
$ curl -O http://example.com/fichier.zip
```

Différence avec wget

`wget` : idéal pour télécharger simplement. `curl` : plus polyvalent (APIs REST, formulaires, authentification. . .).

netstat

- Affiche les **connexions réseau actives** et les **ports ouverts**.
- Utile pour vérifier quels services tournent sur la machine.
- Options courantes :
  - `-t` : connexions TCP
  - `-u` : connexions UDP
  - `-l` : ports en écoute
  - `-p` : affiche le programme associé

```
# Voir les connexions TCP actives
$ netstat -t

# Voir les ports ouverts avec les programmes associés
$ netstat -tulnp
```

Note

Sur les systèmes récents, `ss` remplace `netstat` avec une syntaxe similaire.

nmap

- Scanner réseau permettant de découvrir :
  - les machines actives sur un réseau,
  - les ports ouverts et les services disponibles.
- Très utilisé pour l'administration et l'audit de sécurité.

```
# Scanner les 1000 ports par défaut d'une machine
$ nmap 192.168.1.10

# Scanner tout un réseau local
$ nmap 192.168.1.0/24
```

Attention

`nmap` est un outil puissant : utilisez-le uniquement sur vos propres réseaux ou avec une autorisation.

Configurer le réseau : `ip` vs `ifconfig`

`ip` (paquet `iproute2`) est la commande moderne et complète. **`ifconfig/route`** (paquet `net-tools`) sont historiques et parfois absentes par défaut.

`ip` (recommandé)

- `ip addr` : adresses IP
- `ip link` : interfaces
- `ip route` : table de routage
- `ip -br a` : vue abrégée

`ifconfig/route` (héritage)

- `ifconfig` : interfaces + IP
- `ifconfig -a` : tout afficher
- `route -n` : table de routage

À retenir

Préférer `ip` pour les scripts et l'administration moderne.

Lister et inspecter les interfaces

Avec `ip`

```
# Affichage détaillé
$ ip addr show
$ ip link show

# Vue abrégée (lisible)
$ ip -br addr
$ ip -br link

# Table de routage
$ ip route show
```

Avec `net-tools`

```
# Interfaces (toutes)
$ ifconfig -a

# Interface spécifique
$ ifconfig eth0

# Routage (numérique)
$ route -n
```

Astuce

`ip -br a` fournit un résumé propre (br = *brief*).

Configurer une interface (temporairement)

Avec `ip` (recommandé)

```
# Ajouter une IP (ex.)
$ sudo ip addr add 192.168.1.50/24 dev eth0

# Retirer une IP
$ sudo ip addr del 192.168.1.50/24 dev eth0

# Activer / désactiver l'interface
$ sudo ip link set dev eth0 up
$ sudo ip link set dev eth0 down
```

Avec `ifconfig` (héritage)

```
# Assigner une IP + masque
$ sudo ifconfig eth0 192.168.1.50 \
  netmask 255.255.255.0 up

# Désactiver l'interface
$ sudo ifconfig eth0 down
```

Sécurité en TP

Pour éviter de casser la connectivité, entraînez-vous sur `lo` : `sudo ip addr add 10.10.10.10/32 dev lo` (puis `del` pour retirer).

Routes et passerelle par défaut

Avec `ip`

```
# Voir la route par défaut
$ ip route show

# Ajouter une passerelle par défaut
$ sudo ip route add default \
        via 192.168.1.1 dev eth0

# Supprimer une route
$ sudo ip route del default
```

Avec `route` (héritage)

```
# Afficher la table de routage
$ route -n

# Ajouter une passerelle par défaut
$ sudo route add default \
       gw 192.168.1.1 eth0

# Supprimer la route par défaut
$ sudo route del default
```

Note

Ces modifications sont **temporaires**. Pour les rendre persistantes, utiliser l'outil de la distro (`Netplan`, `NetworkManager`, `/etc/network/interfaces`, fichiers `systemd-networkd`, etc.).

Cheat sheet : équivalents `ip` & `net-tools`

`ip` (iproute2)

- `ip -br addr` — résumé des IP
- `ip addr add/del` — gérer IP
- `ip link set up/down` — activer
- `ip route show/add/del` — routage
- `ip neigh` — cache ARP

`net-tools` (héritage)

- `ifconfig [-a]` — interfaces
- `ifconfig eth0 <ip> netmask <mask>`
- `ifconfig eth0 up/down`
- `route -n, route add/del`
- `arp -n` — table ARP

Conseil

Pour écrire des scripts modernes, **maîtriser `ip`** est indispensable.

Quelques commandes réseau à retenir

- `ping`, `ping -c 4 <hôte>`
- `traceroute`, `tracepath`
- `wget`, `curl`
- `dig <domaine>`
- `arp -n`
- `ifconfig eth0 up/down`
- `nmap <ip>`

- `netstat -tulnp`
- `ss -tulnp`
- `nmap <réseau>/24`
- `ip addr`, `ip -br addr`
- `ip link`, `ip neigh`
- `route -n`, `route add default gw <ip>`

- `ip route show`
- `ip addr add/del`
- `ip link set up/down`
- `ip route add/del default`
- `ifconfig -a`, `ifconfig eth0 <ip>`