Guide Linux pour la semaine 1

Commands

- **`bg`** : Exécute un processus en arrière-plan.
  Exemple : `bg %1` (Reprend le travail `1` en arrière-plan).

- **`fg`** : Ramène un processus suspendu au premier plan.
  Exemple : `fg %1` (Ramène le travail `1` au premier plan).

- **`tee`** : Lit l'entrée et écrit simultanément sur la sortie standard et dans des fichiers.
  Exemple : `echo "Bonjour" | tee fichier.txt` (Écrit "Bonjour" sur le terminal et dans `fichier.txt`).

- **`read`** : Lit l'entrée de l'utilisateur via le terminal.
  Exemple : `read nom` (Demande à l'utilisateur de saisir et stocke la valeur dans `nom`).

- **`chown`** : Change la propriété d'un fichier/répertoire.
  Exemple : `chown utilisateur:groupe fichier.txt` (Change le propriétaire en `utilisateur` et le groupe en `groupe`).

- **`chgrp`** : Change le groupe d'un fichier/répertoire.
  Exemple : `chgrp groupe fichier.txt` (Change le groupe de `fichier.txt` en `groupe`).

- **`chmod`** : Change les permissions d'un fichier (lecture, écriture, exécution).
  Exemple : `chmod 755 fichier.txt` (Définit les permissions `rwxr-xr-x`).

- **`sudo`** : Exécute une commande avec des privilèges superutilisateur.
  Exemple : `sudo apt update` (Exécute la commande en tant que superutilisateur).

- **`ld`** : Lie des fichiers objets (`.o`) pour créer un exécutable.
  Exemple : `ld -o programme fichier1.o fichier2.o` (Lie `fichier1.o` et `fichier2.o` pour créer `programme`).

- **`strace`** : Trace les appels système et signaux d'un programme.
  Exemple : `strace ls` (Affiche tous les appels système effectués par `ls`).

- **`alarm`** : Définit un délai pour un programme ; il est tué lorsque le délai expire.

- **`sed`** : Outil de ligne de commande utilisé pour la manipulation de texte, comme la recherche, le remplacement et la suppression de texte dans un flux ou un fichier.
  Exemple : `sed 's/ancien/nouveau/g' fichier.txt` (Remplace toutes les occurrences de "ancien" par "nouveau" dans `fichier.txt`).

- **`cd`** : Change le répertoire actuel.
  Exemple : `cd /home/utilisateur` (Se déplace dans le répertoire `/home/utilisateur`).

- **`ls`** : Liste le contenu d'un répertoire.
  Exemple : `ls -l` (Liste les fichiers dans le répertoire actuel avec des informations détaillées).

- **`mkdir`** : Crée un nouveau répertoire.
  Exemple : `mkdir nouveau_dossier` (Crée un répertoire nommé `nouveau_dossier`).

- **`cat`** : Affiche le contenu d'un fichier ou concatène plusieurs fichiers.
  Exemple : `cat fichier.txt` (Affiche le contenu de `fichier.txt`).

- **`touch`** : Crée un fichier vide ou met à jour l'horodatage d'un fichier existant.
  Exemple : `touch nouveau_fichier.txt` (Crée un fichier vide `nouveau_fichier.txt` s'il n'existe pas).

- **`gcc`** : Utilisé pour compiler du code assembleur en code binaire, puis en exécutable.
  Syntaxe : `gcc -nostdlib -o <NAME> <FILE>.s`

- **`objdump`** : Désassemble un programme pour visualiser son code machine.
  Syntaxe : `objdump -M <SYNTAX> -d <PROGRAM>`

- **`gdb`** : Utilisé pour le débogage. Permet d'ajouter un point d'arrêt pour interrompre le programme et examiner son état pendant l'exécution.
  Exemple d'instruction de point d'arrêt : `int3`

- **`rsync`** : Un outil rapide et polyvalent pour copier des fichiers et des répertoires localement ou entre systèmes distants. Il transfère uniquement les différences entre la source et la destination, minimisant ainsi le transfert de données.
  Syntaxe : `rsync [options] <source> <destination>`
  Exemple : `rsync -avz /home/user/docs/ user@remote:/backup/docs/`
  Explication : Cette commande synchronise le répertoire `/home/user/docs/` avec le répertoire `/backup/docs/` sur le serveur distant en utilisant les options `-a` (archive), `-v` (verbose) et `-z` (compression).

File Descriptors

- **`f0 - stdin`** : Lit l'entrée depuis le clavier ou un autre programme.
  Exemple : `cat < input.txt` (Redirige `stdin` pour lire à partir de `input.txt`).

- **`f1 - stdout`** : Envoie des données vers le terminal ou un autre programme.
  Exemple : `echo "Bonjour" > output.txt` (Redirige `stdout` vers `output.txt`).

- **`f2 - stderr`** : Envoie les erreurs vers le terminal ou une autre destination.
  Exemple : `ls fichier_inexistant 2> error.log` (Redirige `stderr` vers `error.log`).

File globbing

- `?` : Correspond à **n'importe quel caractère unique**.
  Exemple : `fichier?.txt` → `fichier1.txt`, `fichierA.txt` (pas `fichier10.txt`).
- `[]` : Correspond à **un caractère** dans un ensemble ou une plage.
  Exemple : `fichier[1-3].txt` → `fichier1.txt`, `fichier2.txt`.
  Négation : `[!...]` (par exemple, `fichier[!1-3].txt` → `fichier4.txt`).
- `{}` : Correspond à **des options séparées par des virgules**.
  Exemple : `fichier{1,2,3}.txt` → `fichier1.txt`, `fichier2.txt`.

Piping

- `|` : Passe la sortie d'un programme en entrée d'un autre.
  Exemple : `ls | grep "fichier"` (Liste les fichiers et filtre ceux contenant "fichier").

Processes and Jobs

- **`Ctrl+C`** : Arrête un processus en cours d'exécution.
  Exemple : Appuyez sur `Ctrl+C` pendant l'exécution de `ping google.com` pour l'arrêter.
- **`Ctrl+Z`** : Suspend un processus en cours d'exécution.
  Exemple : Appuyez sur `Ctrl+Z` pendant l'exécution de `nano fichier.txt` pour le suspendre. Utilisez `fg` pour le reprendre.

Chaining Commands

- `;` : Exécute plusieurs commandes de manière séquentielle, indépendamment de leur succès ou échec.
  Exemple : `ls -l /home; cat /data` (Liste le contenu de `/home`, puis affiche le contenu de `/data`).

Special Files

- **`/etc/shadow`** : Contient des informations sensibles sur les utilisateurs (par exemple, mots de passe cryptés).
- **`/etc/sudoers`** : Liste les utilisateurs autorisés à exécuter des commandes avec `sudo`.
- **`/dev/null`** : Un fichier spécial qui rejette toutes les données écrites et renvoie EOF lorsqu'il est lu. Souvent utilisé pour supprimer la sortie.
  Exemple : `commande > /dev/null` (Supprime la sortie standard de `commande`).

Paths

- **Chemin absolu** : Un chemin complet qui commence à partir du répertoire racine (`/`).
  Exemple : `/home/utilisateur/documents/fichier.txt` (Spécifie l'emplacement exact du fichier depuis la racine).

- **Chemin relatif** : Un chemin par rapport au répertoire de travail actuel.
  Exemple : `documents/fichier.txt` (Spécifie le fichier par rapport au répertoire actuel).

- **`$PATH`** : Une variable d'environnement qui stocke une liste de répertoires où les programmes exécutables sont localisés.
  Exemple : `echo $PATH` (Affiche les répertoires listés dans `$PATH` où les commandes sont recherchées).

Assembly concepts

- **`mov`** : L'instruction `mov` est utilisée pour transférer des données entre registres, mémoire et valeurs immédiates. Elle ne réalise pas de calculs mais copie simplement les données.
  Exemple : `mov rax, 10; # Déplace la valeur 10 dans le registre RAX`

- **`syscall`** : L'instruction `syscall` est utilisée pour demander des services au système d'exploitation, comme des opérations sur des fichiers ou la terminaison d'un programme. Elle nécessite que certains registres soient configurés au préalable.
  Exemple : `mov rax, 60; Numéro de l'appel système pour exit mov rdi, 0 ; Code de sortie (0 = succès) syscall ; Termine le programme`

- **`rdi`** : Le registre `rdi` est utilisé pour stocker le premier argument des appels système ou des appels de fonction.
  Exemple : `mov rdi, 1; Définit le descripteur de fichier (1 = stdout)`

- **`rax`** : Le registre `rax` est utilisé pour stocker le numéro de l'appel système.
  Exemple : `mov rax, 1; Définit le numéro de l'appel système pour write`

- **`rsi`** : Le registre `rsi` est utilisé pour stocker le second argument des appels système ou des appels de fonction.
  Exemple : `mov rsi, msg; Définit l'adresse de la chaîne de caractères`

- **Assembly read & write**: Le registre `rdi` est utilisé pour spécifier le descripteur de fichier pour les appels système liés à la lecture et à l'écriture. Mettre `rdi` à `0` indique la lecture depuis l'entrée standard, `1` pour écrire dans la sortie standard, et `2` pour écrire dans l'erreur standard.
  Exemple : `mov rdi, 1; # Met rdi à 1 pour écrire dans la sortie standard`

- **`rdx`** : Le registre `rdx` est utilisé pour représenter le troisième argument d'un appel système. Il est couramment utilisé pour spécifier la taille des données à écrire ou le nombre d'octets à lire.

Exemple : `mov rdx, 13; # Met rdx à 13 pour la taille des données à écrire`

- **`[]`** : Indique au CPU de récupérer la valeur stockée à l'adresse mémoire spécifiée.
  Exemple : `mov rdi, [1234]` (Le CPU récupère la valeur à l'adresse mémoire `1234` et la stocke dans le registre `rdi`)

- **`[addr + n]`** : Indique au CPU de récupérer les données à partir d'une adresse mémoire spécifique avec un décalage, souvent utilisé pour accéder à des éléments d'une collection ou d'un tableau.
  Exemple : `mov rsi, [rax + 4]` (Le CPU récupère la valeur stockée à l'adresse contenue dans `rax` plus 4 octets et la stocke dans le registre `rsi`)

- **`.intel_syntax`** : Définit la syntaxe à utiliser pour les codes assembleurs.
  Syntaxe : `.intel_syntax noprefix`

- **`.global _start`** : Définit la variable `_start`.

- **`_start`** : Définit le point de départ des codes assembleurs.

- **`add`** : Ajoute la valeur d'un registre à un autre.
  Syntaxe : `add reg1, reg2 <=> reg1 += reg2`

- **`imul`** : Multiplie (avec signe) la valeur d'un registre par un autre.
  Syntaxe : `imul reg1, reg2 <=> reg1 *= reg2`

- **`div`** : Divise les 64 bits supérieurs d'un dividende de 128 bits par les 64 bits inférieurs.
  Syntaxe : `mov rax, rdi; div rsi`
  Explication : Le dividende est placé dans `rax` (à partir de `rdi`), et `rsi` est le diviseur. Après la division, le quotient est stocké dans `rax` et le reste dans `rdx`.

- **`xor`** : Efface un registre en le XORant avec lui-même.
  Syntaxe : `xor rdx, rdx`

- **`eax`** : Accède aux 32 bits inférieurs de `rax`.

- **`ax - bx`** : Accède aux 16 bits inférieurs de `rax`.

- **`ah - al`** : Représente les 8 bits inférieurs de `ax`.

- **`dil`** : Accède aux 8 bits inférieurs de `rdi`.

- **`si`** : Accède aux 16 bits inférieurs de `rsi`.

Shell Script Concepts

- **`$@`** : Représente tous les arguments passés au script ou à la fonction. Chaque argument est traité comme une entité distincte.
  Exemple : `echo $@; # Affiche tous les arguments passés au script`

- **`$#`** : Représente le nombre d'arguments passés au script ou à la fonction.
  Exemple : `echo $#; # Affiche le nombre d'arguments passés au script`

- **`$1, $2, ...`** : Représente les arguments individuels passés au script ou à la fonction. `$1` est le premier argument, `$2` est le deuxième, et ainsi de suite.
  Exemple : `echo $1; # Affiche le premier argument passé au script`