V 1.0

Lexique général pour ./minishell

Auteurs :

Viktorija JANKOVSKAJA
Nikita VASILEV

TABLE DES MATIERES

1. Caractériques d’un ordinateur 2

1.1. Qu’est qu’un ordinateur ? 2

1.2. Comment est exécuté un programme ? 4

1.3. Qu’est-ce qu’un CPU ? 5

1.4. Comment fonctionne un CPU ? 5

1.5. Quelles sont les différents types d'architectures ? 7

1.6. Qu'est-ce que la RAM ? 8

2. Systèmes d’exploitation, kernel, POSIX et fonctionnement 9

2.1. Qu'est-ce qu'un système d'exploitation et un kernel ? 9

2.2. Quelle est la différence entre UNIX, Linux, GNU-Linux et BSD ? 10

2.3. Et macOS dans tout ça ? 11

2.4. POSIX c'est quoi ? 12

2.5. Un appel système, c’est quoi ? 12

2.6. Qu'est-ce qu'un processus ? 13

2.7. Et les processus enfants dans tout ça ? 15

2.8. Qu'est-ce qu'un signal UNIX ? 16

2.9. Qu'est-ce qu'un file descriptor ? 16

2.10. Streams vs. File Descriptors 17

3. Command Line Interface 18

3.1. Qu’est-ce qu’un shell ? 18

3.2. Bash (Bourne-Again SHell), c’est quoi ? 19

3.3. Implémentation de Bash 19

3.4. Commandes, exécutables et builtins 22

3.5. Environnement d'un shell 22

3.6. Qu'est-ce qu'un chemin relatif et un chemin absolu (Unix) ? 23

3.7. Pourquoi exécute-t-on une commande dans un processus enfant ? 24

3.8. Les redirections c’est quoi ? 25

1. Caractériques d’un ordinateur

1.1. Qu’est qu’un ordinateur ?

Un ordinateur est une machine complexe conçue pour traiter et exécuter des instructions informatiques. Il est composé de plusieurs composants matériels et logiciels qui travaillent ensemble pour réaliser des tâches et des opérations spécifiques. Voici une explication simplifiée des principaux composants d'un ordinateur et de leur fonctionnement :

1. **CPU (Central Processing Unit)** : Le CPU est souvent considéré comme le "cerveau" de l'ordinateur. Il exécute les instructions du programme stockées en mémoire et effectue les calculs nécessaires. Les CPU sont composés de plusieurs cœurs, chacun pouvant traiter des instructions séparément, ce qui permet de traiter plusieurs tâches simultanément (multitâche).

2. **GPU (Graphics Processing Unit)** : Le GPU est spécialisé dans le traitement de l'affichage graphique et des images. Contrairement au CPU, qui est conçu pour être polyvalent, le GPU est optimisé pour traiter rapidement un grand nombre d'opérations mathématiques parallèles, ce qui est essentiel pour générer des graphiques complexes et des animations en temps réel. Les GPU sont également utilisés pour des tâches de calcul parallèle, comme l'intelligence artificielle et le deep learning.

3. **RAM (Random Access Memory)** : La RAM est la mémoire à court terme de l'ordinateur. Elle stocke temporairement les données et les instructions qui sont utilisées par le CPU et le GPU. La RAM est volatile, ce qui signifie qu'elle perd toutes les données qu'elle contient lorsque l'ordinateur est éteint. Une RAM plus grande permet à l'ordinateur de stocker davantage de données pour un accès rapide, ce qui peut améliorer les performances globales.

4. **Stockage** : Les disques durs (HDD) et les disques à semi-conducteurs (SSD) sont les principaux dispositifs de stockage dans un ordinateur. Ils conservent les données de manière permanente, même lorsque l'ordinateur est éteint. Les SSD sont généralement plus rapides que les HDD, mais peuvent être plus coûteux en termes de coût par gigaoctet.

5. **Carte mère** : La carte mère est le composant principal qui relie tous les autres composants de l'ordinateur. Elle assure la communication entre le CPU, le GPU, la RAM, les dispositifs de stockage et les périphériques externes. La carte mère contient également le BIOS (Basic Input/Output System), un logiciel de bas niveau qui gère les paramètres matériels de base et assure le démarrage de l'ordinateur.

6. **Périphériques d'entrée et de sortie** : Les périphériques d'entrée, tels que les claviers, les souris et les webcams, permettent aux utilisateurs de communiquer avec l'ordinateur en fournissant des données et des commandes. Les périphériques de sortie, tels que les moniteurs, les imprimantes et les haut-parleurs, permettent à l'ordinateur de présenter des informations aux utilisateurs sous forme visuelle, auditive ou imprimée.

7. **Alimentation électrique** : L'alimentation électrique fournit l'énergie nécessaire au fonctionnement de tous les composants de l'ordinateur. Elle convertit le courant alternatif (AC) du réseau électrique en courant continu (DC) à différentes tensions, selon les besoins des différents composants.

8. **Refroidissement** : Les composants d'un ordinateur, en particulier le CPU et le GPU, peuvent générer beaucoup de chaleur lorsqu'ils fonctionnent. Les systèmes de refroidissement, tels que les dissipateurs thermiques, les ventilateurs et les systèmes de refroidissement par liquide, sont utilisés pour dissiper cette chaleur et maintenir les températures des composants dans des limites sûres et optimales.

9. **Système d'exploitation** : Le système d'exploitation (OS) est un logiciel essentiel qui permet aux utilisateurs et aux applications d'interagir avec le matériel de l'ordinateur. Il gère les ressources matérielles, les fichiers, les processus et les périphériques, et fournit une interface utilisateur pour faciliter l'utilisation de l'ordinateur. Des exemples de systèmes d'exploitation courants incluent Microsoft Windows, macOS, Linux et Android.

10. **Logiciels** : Les logiciels sont des programmes informatiques conçus pour effectuer des tâches spécifiques. Ils sont généralement classés en logiciels applicatifs, qui permettent aux utilisateurs d'effectuer des tâches spécifiques (comme la navigation sur le Web, la rédaction de documents ou la retouche photo), et en logiciels système, qui gèrent et contrôlent le matériel et les autres logiciels.

En résumé, un ordinateur est une machine complexe composée de nombreux composants matériels et logiciels interagissant les uns avec les autres pour effectuer diverses tâches. Les éléments clés d'un ordinateur incluent le CPU, la mémoire, le stockage, la carte mère, les périphériques d'entrée et de sortie, le système d'exploitation, les logiciels, l'alimentation électrique et les systèmes de refroidissement. Chacun de ces composants joue un rôle essentiel dans le fonctionnement général de l'ordinateur, et leur intégration et leur coordination permettent aux utilisateurs d'accomplir des tâches complexes et variées avec une facilité et une efficacité croissantes.

1.2. Comment est exécuté un programme ?

L'exécution d'un programme suit plusieurs étapes. Voici un aperçu général de ces étapes pour vous donner une idée de comment un programme est exécuté :

1. **Écriture du code source** : Le développeur écrit le code source en utilisant un langage de programmation, tel que Python, Java, C++, etc. Le code source est une série d'instructions destinées à être exécutées par l'ordinateur.

2. **Compilation ou interprétation** : Selon le langage de programmation utilisé, le code source est soit compilé en un fichier exécutable (dans le cas des langages compilés, comme C++), soit interprété directement (dans le cas des langages interprétés, comme Python).

    - Compilation : Le compilateur traduit le code source en un fichier exécutable, généralement en code binaire ou en bytecode, qui peut être directement exécuté par l'ordinateur. Ce fichier contient des instructions spécifiques au processeur et au système d'exploitation cible.

    - Interprétation : L'interpréteur lit et exécute le code source ligne par ligne. Les instructions sont traduites en opérations pour le processeur au fur et à mesure de leur lecture, sans créer de fichier exécutable séparé.

3. **Chargement en mémoire** : Lorsque le programme est lancé, le système d'exploitation charge le fichier exécutable ou le code source interprété en mémoire. Il alloue également de la mémoire pour les variables et les structures de données utilisées par le programme.

4. **Exécution** : Le processeur commence à exécuter les instructions du programme, une à la fois, en suivant un ordre séquentiel. Cependant, en fonction des instructions de contrôle de flux (telles que les boucles et les conditions), l'ordre d'exécution des instructions peut changer. Les instructions sont exécutées sous la supervision du système d'exploitation, qui gère l'accès aux ressources matérielles et logicielles.

5. **Entrées/Sorties** : Pendant l'exécution du programme, il peut interagir avec d'autres systèmes ou avec l'utilisateur en effectuant des opérations d'entrée/sortie (I/O). Ces opérations peuvent inclure la lecture et l'écriture de fichiers, l'affichage de données à l'écran ou la collecte d'informations via un clavier, une souris ou d'autres dispositifs d'entrée.

6. **Gestion des erreurs** : En cours d'exécution, le programme peut rencontrer des erreurs ou des exceptions, telles que des erreurs de syntaxe, des erreurs logiques, des erreurs d'exécution ou des problèmes liés aux ressources matérielles. Le programme peut contenir du code pour gérer ces erreurs ou exceptions de manière appropriée, ou bien

les laisser remonter au système d'exploitation qui peut les traiter ou les signaler à l'utilisateur.

7. **Libération des ressources** : Lorsque le programme se termine, il libère les ressources qu'il a utilisées, telles que la mémoire allouée, les fichiers ouverts, les connexions réseau, etc. Le système d'exploitation peut également effectuer un nettoyage supplémentaire pour s'assurer que toutes les ressources ont été correctement libérées.

8. **Terminaison** : Enfin, le programme se termine, et le contrôle est rendu au système d'exploitation ou au processus parent qui a lancé le programme.

C'est un aperçu simplifié des étapes impliquées dans l'exécution d'un programme. Les détails peuvent varier en fonction du langage de programmation, du système d'exploitation et de l'architecture matérielle utilisée.

1.3. Qu’est-ce qu’un CPU ?

Un CPU (Central Processing Unit), ou unité centrale de traitement en français, est un composant essentiel d'un ordinateur et d'autres dispositifs électroniques programmables. Il est souvent considéré comme le "cerveau" de l'ordinateur, car il est responsable de l'exécution des instructions et du traitement des données.

Le CPU effectue des opérations de calcul et de logique, gère les entrées et les sorties, et exécute des instructions provenant de la mémoire. Il est composé de plusieurs éléments, tels que l'unité arithmétique et logique (ALU), qui effectue les opérations mathématiques et logiques, et l'unité de contrôle, qui gère les instructions et les données en transit.

Les performances d'un CPU sont mesurées en termes de vitesse d'horloge (exprimée en gigahertz, GHz), de nombre de cœurs (qui permettent un traitement parallèle des tâches), et de taille de la mémoire cache (qui influe sur la rapidité d'accès aux données fréquemment utilisées).

Les CPU sont fabriqués par diverses entreprises, dont Intel et AMD, qui sont parmi les plus grands fabricants de CPU pour les ordinateurs de bureau et les serveurs. D'autres fabricants, comme ARM, conçoivent des CPU pour les appareils mobiles et les systèmes embarqués.

1.4. Comment fonctionne un CPU ?

Le fonctionnement d'un CPU peut être expliqué de manière simplifiée en quelques étapes principales :

1. **Chargement des instructions** : Les instructions, qui sont les tâches à effectuer par le CPU, sont stockées dans la mémoire de l'ordinateur. Le CPU récupère ces instructions à partir de la mémoire en utilisant un registre spécial appelé "compteur de programme" (PC, pour Program Counter) qui indique l'emplacement de l'instruction actuelle.

2. **Décodage des instructions** : Une fois l'instruction chargée, elle est décodée par l'unité de contrôle du CPU pour déterminer quelles opérations doivent être effectuées et quels registres ou emplacements mémoire sont impliqués.

3. **Exécution des instructions** : L'unité arithmétique et logique (ALU) du CPU effectue les opérations spécifiées par l'instruction décodée, telles que les calculs mathématiques, les opérations logiques ou les manipulations de données.

4. **Stockage des résultats** : Les résultats de l'exécution sont stockés dans les registres internes du CPU ou dans la mémoire de l'ordinateur, selon les instructions.

5. **Répétition du processus** : Le compteur de programme est mis à jour pour pointer vers la prochaine instruction, et le processus se répète jusqu'à ce que l'ordinateur soit éteint ou qu'une instruction d'arrêt soit exécutée.

Ces étapes sont effectuées en continu, à une vitesse très rapide, généralement plusieurs milliards de fois par seconde, permettant au CPU de traiter un grand nombre d'instructions et de tâches en peu de temps. La vitesse d'exécution des instructions dépend de la vitesse d'horloge du CPU, qui est mesurée en gigahertz (GHz). Un CPU avec une vitesse d'horloge plus élevée peut traiter plus d'instructions par seconde, ce qui se traduit généralement par de meilleures performances.

Il est important de noter que, bien que le fonctionnement d'un CPU soit simplifié ici, il existe une grande variété de CPU avec des architectures et des fonctionnalités différentes. Certaines conceptions de CPU incluent plusieurs cœurs, permettant un traitement parallèle des instructions et une meilleure gestion des tâches simultanées.

De plus, les CPU modernes sont équipés de fonctionnalités avancées telles que la prédiction de branches, qui tente d'anticiper les instructions futures et d'améliorer la vitesse d'exécution en pré chargeant les instructions et les données pertinentes, et l'hyperthreading, qui permet à un seul cœur d'exécuter plusieurs threads simultanément, améliorant ainsi l'efficacité globale.

En résumé, un CPU fonctionne en chargeant, décodant et exécutant des instructions stockées dans la mémoire de l'ordinateur, puis en stockant les résultats. Ce processus se répète à une vitesse très rapide, permettant au CPU de gérer de nombreuses tâches et de fournir les performances nécessaires pour faire fonctionner les systèmes informatiques modernes.

1.5. Quelles sont les différents types d'architectures ?

Il existe plusieurs architectures de processeurs, chacune avec ses propres caractéristiques et avantages. Les architectures les plus courantes sont x86, x86-64 (également appelée AMD64 ou Intel 64) et ARM. Voici une brève description de chacune :

1. **x86** : x86 est une architecture de processeur développée par Intel dans les années 1970 et largement adoptée dans les ordinateurs personnels (PC) depuis lors. Cette architecture est basée sur un ensemble d'instructions complexe (CISC) et supporte principalement les systèmes d'exploitation 32 bits. Les processeurs x86 ont été les principaux choix pour les ordinateurs de bureau et les ordinateurs portables pendant des décennies, bien qu'ils aient progressivement été remplacés par des processeurs x86-64.

2. **x86-64 (AMD64 ou Intel 64)** : x86-64 est une extension de l'architecture x86 qui ajoute la prise en charge des systèmes d'exploitation 64 bits. Cette architecture a été introduite par AMD en 2000 sous le nom d'AMD64 et adoptée plus tard par Intel sous le nom d'Intel 64. Les processeurs x86-64 sont rétrocompatibles avec les instructions x86, ce qui signifie qu'ils peuvent exécuter des applications 32 bits et 64 bits. L'avantage principal de l'architecture 64 bits est qu'elle permet au processeur d'accéder à plus de mémoire vive (RAM) et de traiter de plus grandes quantités de données en une seule opération.

3. **ARM** : ARM (Advanced RISC Machine) est une architecture de processeur basée sur un ensemble d'instructions réduit (RISC) et conçue pour une faible consommation d'énergie et une haute efficacité énergétique. Les processeurs ARM sont largement utilisés dans les appareils mobiles, les systèmes embarqués et l'Internet des objets (IoT) en raison de leur faible consommation d'énergie et de leur taille réduite. Ils sont également de plus en plus utilisés dans les serveurs et les ordinateurs portables en raison de leur efficacité énergétique et de leur capacité à offrir des performances compétitives par rapport aux processeurs x86 et x86-64.

   Les processeurs ARM sont disponibles en versions 32 bits (ARMv7 et antérieurs) et 64 bits (ARMv8 et ultérieurs). L'architecture ARM 64 bits permet, comme x86-64, d'accéder à plus de mémoire vive et de traiter de plus grandes quantités de données en une seule opération par rapport à son homologue 32 bits.

En résumé, les architectures x86, x86-64 et ARM sont les principales architectures de processeurs utilisées dans les ordinateurs et les appareils électroniques. Les processeurs x86 et x86-64 sont couramment utilisés dans les ordinateurs de bureau et les serveurs, tandis que les processeurs ARM sont largement utilisés dans les appareils mobiles et les systèmes embarqués. Les architectures 32 bits et 64 bits se réfèrent à la taille des registres et à la quantité de mémoire vive qu'un processeur peut gérer, avec les versions 64 bits offrant une capacité de traitement des données et d'accès à la mémoire plus importante que les versions 32 bits.

1.6. Qu'est-ce que la RAM ?

La RAM (Random Access Memory, ou mémoire vive en français) est un type de mémoire utilisé dans les ordinateurs et autres dispositifs électroniques pour stocker temporairement des données et des instructions nécessaires au fonctionnement du système. Contrairement à la mémoire de stockage (comme les disques durs ou les SSD), la RAM est volatile, ce qui signifie qu'elle perd son contenu lorsque l'appareil est éteint.

La RAM joue un rôle crucial dans la performance globale d'un système informatique, car elle permet un accès rapide aux données fréquemment utilisées et aux instructions en cours d'exécution. Plus la capacité de la RAM est importante, plus le système peut gérer simultanément un grand nombre de tâches et d'applications sans ralentir.

La RAM n'est pas directement composée de la pile (stack) et du tas (heap), mais ces deux éléments sont des parties de la mémoire gérées par le système d'exploitation et utilisées pour stocker des données durant l'exécution des programmes. Voici une description de chaque élément :

1. **Stack (Pile)** : La pile est une région de la mémoire organisée selon le principe "dernier arrivé, premier sorti" (LIFO). Elle est principalement utilisée pour stocker des données temporaires, telles que les variables locales et les informations de retour des fonctions. La pile est gérée automatiquement par le système d'exploitation et elle se développe et se réduit à mesure que les fonctions sont appelées et retournées.

2. **Heap (Tas)** : Le tas est une région de la mémoire utilisée pour allouer dynamiquement de l'espace mémoire pendant l'exécution des programmes. Contrairement à la pile, le tas n'est pas géré automatiquement et les programmeurs sont responsables de l'allocation et de la libération de la mémoire utilisée. Les objets et les données créés dans le tas ont une durée de vie plus longue que ceux stockés dans la pile et peuvent être utilisés pendant toute la durée de vie du programme. Cependant, une mauvaise gestion de la mémoire du tas peut entraîner des fuites de mémoire et des problèmes de performance.

3. **Text (Code)** : La section Text, également appelée section Code, contient le code binaire du programme en cours d'exécution. Elle comprend les instructions du processeur et les routines de gestion des exceptions. Cette section est généralement en lecture seule, ce qui signifie qu'elle ne peut pas être modifiée pendant l'exécution du programme. L'objectif de cette restriction est de protéger le code du programme contre les modifications accidentelles ou malveillantes.

4. **Data** : La section Data est utilisée pour stocker les variables globales et statiques du programme. Cette section est divisée en deux sous-sections : les données initialisées (Initialized Data) et les données non initialisées (Uninitialized Data, aussi appelée BSS pour Block Started by Symbol). Les données initialisées contiennent les valeurs des

variables globales et statiques définies par le programmeur, tandis que les données non initialisées sont utilisées pour les variables globales et statiques qui ne sont pas explicitement initialisées par le programmeur et sont automatiquement initialisées à zéro par le système d'exploitation.

En résumé, la RAM est un type de mémoire volatile utilisée pour stocker temporairement des données et des instructions nécessaires au fonctionnement des systèmes informatiques. La pile et le tas sont des régions de la mémoire gérées par le système d'exploitation et utilisées pour stocker des données pendant l'exécution des programmes. La pile est utilisée pour stocker des données temporaires et est gérée automatiquement, tandis que le tas est utilisé pour allouer dynamiquement de l'espace mémoire et nécessite une gestion manuelle par les programmeurs. La section Text contient le code binaire du programme, et la section Data est utilisée pour stocker les variables globales et statiques, qu'elles soient initialisées ou non.

Pour plus de détails : https://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/memory.html

2. Systèmes d'exploitation, kernel, POSIX et fonctionnement

2.1. Qu'est-ce qu'un système d'exploitation et un kernel ?

Un système d'exploitation (OS, pour Operating System) est un logiciel qui gère les ressources matérielles et logicielles d'un ordinateur ou d'un autre dispositif électronique. Il sert d'intermédiaire entre les applications et le matériel, permettant aux utilisateurs et aux programmes d'interagir avec les composants matériels, tels que le processeur, la mémoire, les dispositifs de stockage et les périphériques d'entrée/sortie. Les systèmes d'exploitation les plus courants sont Microsoft Windows, macOS, Linux et Android.

Le noyau (kernel) est le cœur du système d'exploitation et constitue la partie la plus essentielle de celui-ci. Il est responsable de la gestion des ressources matérielles, du contrôle des processus, de la gestion de la mémoire et de la communication entre les composants matériels et logiciels. Le noyau fonctionne à un niveau très bas, ce qui lui permet de contrôler et de surveiller directement le matériel de l'ordinateur.

Les principales fonctions du noyau incluent :

1. **Gestion des processus** : Le noyau gère la création, l'exécution, la suspension et la terminaison des processus. Il attribue des ressources aux processus et veille à ce qu'ils s'exécutent sans interférer les uns avec les autres.

2. **Gestion de la mémoire** : Le noyau est responsable de l'allocation et de la libération de la mémoire pour les processus en cours d'exécution. Il gère également la mémoire virtuelle et assure l'isolation de la mémoire entre les processus pour éviter qu'ils n'accèdent à la mémoire des autres processus.

3. **Gestion des périphériques** : Le noyau gère les communications entre les applications et les périphériques matériels tels que les disques durs, les écrans, les claviers et les souris. Il utilise des pilotes de périphériques pour interagir avec les différents matériels.

4. **Gestion des fichiers** : Le noyau gère les systèmes de fichiers et fournit des fonctions pour créer, lire, écrire et supprimer des fichiers et des répertoires.

En résumé, un système d'exploitation est un logiciel qui gère les ressources matérielles et logicielles d'un ordinateur et permet aux utilisateurs et aux programmes d'interagir avec le matériel. Le noyau est le cœur du système d'exploitation et est responsable de la gestion des ressources matérielles, du contrôle des processus, de la gestion de la mémoire, de la communication entre les composants matériels et logiciels, ainsi que de la gestion des fichiers et des périphériques. Les systèmes d'exploitation les plus courants incluent Microsoft Windows, macOS, Linux et Android, et jouent un rôle essentiel dans le fonctionnement quotidien des ordinateurs et autres dispositifs électroniques.

2.2. Quelle est la différence entre UNIX, Linux, GNU-Linux et BSD ?

UNIX, Linux, GNU-Linux et BSD sont tous des systèmes d'exploitation, mais ils ont des origines, des objectifs et des caractéristiques distinctes. Voici un aperçu des différences entre ces systèmes d'exploitation :

1. **UNIX** : UNIX est un système d'exploitation développé dans les années 1960 et 1970 par AT&T Bell Labs. Il est conçu pour être portable, multi-utilisateur et multitâche. UNIX est un système d'exploitation propriétaire, ce qui signifie qu'il est soumis à des restrictions de licence et n'est pas librement accessible ni modifiable par le public. Les variantes d'UNIX incluent Solaris, AIX et HP-UX.

2. **Linux** : Linux est un système d'exploitation basé sur UNIX et développé par Linus Torvalds en 1991. Contrairement à UNIX, Linux est un projet open source, ce qui signifie que son code source est accessible au public et que quiconque peut le modifier et le redistribuer. Linux est le noyau du système d'exploitation, qui gère les ressources matérielles et les processus.

3. **GNU-Linux** : GNU-Linux est un système d'exploitation composé du noyau Linux et des outils et bibliothèques du projet GNU. Le projet GNU a été initié par Richard Stallman en 1983 dans le but de créer un système d'exploitation entièrement libre et compatible avec

UNIX. Cependant, le projet GNU n'a pas réussi à développer un noyau fonctionnel (Hurd). Lorsque le noyau Linux est apparu, il a été combiné avec les outils GNU pour créer un système d'exploitation complet et libre. Aujourd'hui, la plupart des "distributions Linux" sont en réalité des systèmes GNU-Linux, comme Ubuntu, Debian, Fedora, etc.

4. **BSD** : BSD (Berkeley Software Distribution) est une famille de systèmes d'exploitation dérivés du système UNIX, développée à l'Université de Californie à Berkeley dans les années 1970 et 1980. BSD est également un projet open source, et certaines de ses variantes les plus populaires incluent FreeBSD, NetBSD et OpenBSD. Les systèmes BSD partagent de nombreuses similitudes avec UNIX et Linux, mais ils ont leur propre base de code et une licence différente (la licence BSD).

En résumé, UNIX est un système d'exploitation propriétaire à l'origine de nombreux autres systèmes d'exploitation. Linux est un noyau de système d'exploitation open source basé sur UNIX, tandis que GNU-Linux est un système d'exploitation complet composé du noyau Linux et des outils GNU. BSD est une famille de systèmes d'exploitation dérivés d'UNIX avec sa propre base de code et licence.

2.3. Et macOS dans tout ça ?

macOS est le système d'exploitation développé par Apple pour ses ordinateurs Mac. Il a été conçu pour offrir une expérience utilisateur élégante et intégrée, avec une interface utilisateur graphique attrayante et une suite d'applications propriétaires. macOS est basé sur un autre système d'exploitation appelé Darwin, qui à son tour a des racines dans BSD, un système d'exploitation dérivé d'UNIX.

En 2000, Apple a sorti Mac OS X, une refonte majeure de son système d'exploitation qui s'appuyait sur Darwin. Darwin est un système d'exploitation open source qui combine des éléments de FreeBSD (un membre de la famille BSD) et de la propre technologie d'Apple, telle que le noyau XNU. Ainsi, macOS est un descendant d'UNIX, et il est conforme aux normes POSIX, tout comme UNIX, Linux et BSD.

Cependant, macOS est un système d'exploitation propriétaire, ce qui signifie qu'Apple contrôle étroitement son code source et sa distribution. Bien que certaines parties de Darwin soient open source, les éléments spécifiques à macOS, tels que l'interface utilisateur, les applications propriétaires et les pilotes de périphériques, ne sont pas librement accessibles ni modifiables.

En résumé, macOS est un système d'exploitation développé par Apple pour ses ordinateurs Mac, basé sur Darwin, qui a des racines dans la famille BSD et, par conséquent, dans UNIX. Bien qu'il partage certaines similitudes avec UNIX, Linux et BSD, macOS est un système d'exploitation propriétaire avec une interface utilisateur et des applications spécifiques à Apple.

2.4. POSIX c'est quoi ?

POSIX (Portable Operating System Interface) est une famille de normes définies par l'Institute of Electrical and Electronics Engineers (IEEE) pour garantir la compatibilité et la portabilité entre les différents systèmes d'exploitation. L'objectif de POSIX est de permettre aux développeurs d'écrire des programmes qui peuvent être exécutés sur divers systèmes d'exploitation sans nécessiter de modifications importantes du code.

Les normes POSIX définissent un ensemble d'interfaces, d'environnements de programmation et d'exigences de comportement pour les systèmes d'exploitation. Elles couvrent des aspects tels que la gestion des processus, les signaux, les fichiers, les répertoires, les entrées/sorties, la gestion de la mémoire et les communications inter-processus. Les normes POSIX spécifient également les fonctions de bibliothèque et les appels système utilisés pour interagir avec le système d'exploitation.

Les systèmes d'exploitation conformes à POSIX incluent UNIX, Linux, BSD et macOS. Ces systèmes partagent un ensemble commun d'interfaces et de comportements, ce qui facilite la portabilité des logiciels entre eux. Il est important de noter que la conformité POSIX ne garantit pas une compatibilité parfaite entre les systèmes, car chaque système d'exploitation peut implémenter des fonctionnalités supplémentaires ou se comporter légèrement différemment dans certaines situations.

En résumé, POSIX est une famille de normes visant à garantir la compatibilité et la portabilité entre les systèmes d'exploitation. Les normes POSIX définissent un ensemble commun d'interfaces et de comportements pour les systèmes d'exploitation, ce qui facilite la portabilité des logiciels entre eux. Les systèmes conformes à POSIX, tels qu'UNIX, Linux, BSD et macOS, partagent un ensemble commun de fonctions et d'appels système, ce qui permet aux développeurs de créer des applications qui peuvent être exécutées sur différents systèmes avec un minimum de modifications. Cependant, il est important de garder à l'esprit que la conformité POSIX ne garantit pas une compatibilité parfaite entre les systèmes, car ils peuvent implémenter des fonctionnalités supplémentaires ou se comporter légèrement différemment dans certaines situations.

2.5. Un appel système, c’est quoi ?

Un appel système (ou syscall) est une interface entre le code d'une application et les services fournis par le noyau d'un système d'exploitation. Les appels système permettent aux programmes d'accéder aux ressources système et d'effectuer des opérations de bas niveau, telles que la gestion des fichiers, la communication réseau, la gestion de la mémoire et la création de processus.

Lorsqu'un programme a besoin d'accéder à une ressource système ou d'exécuter une opération de bas niveau, il effectue un appel système en demandant au noyau d'exécuter cette opération en son nom. Le noyau vérifie alors si le programme dispose des autorisations appropriées et, si tel est le cas, effectue l'opération demandée et renvoie les résultats au programme.

Les appels système sont importants pour la sécurité et la stabilité des systèmes informatiques, car ils permettent au noyau de contrôler l'accès aux ressources matérielles et de prévenir les interférences entre les programmes. En isolant les programmes du matériel et en les obligeant à passer par le noyau pour effectuer des opérations de bas niveau, les appels système contribuent à protéger les données et à assurer le bon fonctionnement des différents composants du système.

Voici quelques exemples courants d'appels système :

- read() : Lire des données à partir d'un fichier ou d'un autre périphérique d'entrée.
- write() : Écrire des données dans un fichier ou vers un autre périphérique de sortie.
- open() : Ouvrir un fichier en vue de sa lecture ou de son écriture.
- close() : Fermer un fichier précédemment ouvert.
- fork() : Créer un nouveau processus en copiant le processus actuel.
- exec() : Remplacer l'image mémoire d'un processus par celle d'un nouveau programme.
- wait() : Attendre la fin d'un processus enfant.

En résumé, un appel système est une interface entre le code d'une application et les services fournis par le noyau d'un système d'exploitation. Les appels système permettent aux programmes d'accéder aux ressources système et d'effectuer des opérations de bas niveau sous le contrôle du noyau, contribuant ainsi à la sécurité et à la stabilité du système.

2.6. Qu'est-ce qu'un processus ?

Un processus est une instance d'un programme en cours d'exécution sur un système informatique. Il s'agit d'une entité indépendante qui dispose de sa propre mémoire, de ses propres ressources et de son propre état. Chaque processus est géré par le système d'exploitation, qui alloue les ressources nécessaires, tels que le temps processeur et la mémoire, et coordonne l'exécution du processus.

Un processus est constitué des éléments suivants :

- **Code exécutable** : Il s'agit du programme lui-même, généralement stocké sous forme de fichier binaire sur disque et chargé en mémoire lors de l'exécution.

- **Espace mémoire** : Chaque processus dispose de son propre espace mémoire isolé, qui contient le code exécutable, les données, la pile et le tas. L'isolation de la mémoire entre

les processus empêche qu'un processus n'accède ou ne modifie la mémoire d'un autre processus.

- **Contexte d'exécution** : Il s'agit de l'ensemble des informations nécessaires pour exécuter un processus, telles que les registres du processeur, le pointeur d'instruction et les variables d'environnement.

- **Descripteurs de fichiers et autres ressources** : Les processus peuvent ouvrir et manipuler des fichiers, des sockets réseau et d'autres ressources système. Les descripteurs de fichiers permettent au processus d'interagir avec ces ressources.

- **État du processus** : Un processus peut se trouver dans différents états, tels que prêt (en attente d'exécution), en cours d'exécution ou bloqué (en attente d'un événement, comme la fin d'une opération d'entrée/sortie).

Le système d'exploitation est responsable de la gestion des processus. Il crée, planifie, suspend et termine les processus, en veillant à ce qu'ils s'exécutent correctement et sans interférer les uns avec les autres. Les systèmes d'exploitation modernes permettent l'exécution simultanée de plusieurs processus (multitâche), en utilisant des techniques de planification et de partage du temps processeur pour donner l'impression que les processus s'exécutent en parallèle.

En résumé, un processus est une instance d'un programme en cours d'exécution, avec son propre espace mémoire, contexte d'exécution, descripteurs de fichiers, ressources et état. Le système d'exploitation est responsable de la gestion des processus, de l'allocation des ressources et de la coordination de leur exécution.

Les processus peuvent communiquer entre eux et synchroniser leurs actions par le biais de mécanismes de communication interprocessus (IPC), tels que les files d'attente de messages, les tubes (pipes), les sémaphores, les mémoires partagées et les signaux. Ces mécanismes permettent aux processus de partager des informations, de coordonner leurs actions et de coopérer pour accomplir des tâches complexes.

Il est important de noter que les processus sont différents des threads, bien qu'ils partagent certaines similitudes. Les threads sont des unités d'exécution plus légères qui font partie d'un processus et partagent le même espace mémoire et les mêmes ressources avec d'autres threads du même processus. Les threads permettent l'exécution simultanée de plusieurs tâches au sein d'un même processus, ce qui peut améliorer l'efficacité et les performances d'un programme.

Enfin, la gestion des processus est un aspect essentiel de la sécurité des systèmes d'exploitation. Les systèmes d'exploitation utilisent des mécanismes tels que les permissions d'accès, l'isolation de la mémoire et les espaces d'adressage indépendants pour protéger les processus et les ressources système contre les accès non autorisés et les interférences

malveillantes. Ces protections permettent de garantir la stabilité, la fiabilité et la confidentialité des informations et des processus sur un système informatique.

2.7. Et les processus enfants dans tout ça ?

Les processus enfants, également appelés processus fils, sont des processus qui sont créés par un autre processus, appelé processus parent. La création d'un processus enfant se fait généralement par le biais d'une opération de clonage ou de fork.

Lorsqu'un processus parent crée un processus enfant, le nouveau processus hérite de certaines propriétés du processus parent, telles que les descripteurs de fichiers ouverts, les variables d'environnement et les permissions d'accès aux ressources. Cependant, le processus enfant possède son propre espace mémoire, son propre contexte d'exécution et son propre état, indépendants de ceux du processus parent.

Après la création d'un processus enfant, le processus parent et le processus enfant s'exécutent simultanément et indépendamment l'un de l'autre. Ils peuvent communiquer et synchroniser leurs actions à l'aide des mécanismes de communication interprocessus (IPC) mentionnés précédemment.

Les processus enfants sont souvent utilisés pour exécuter des tâches spécifiques ou pour créer des processus parallèles qui travaillent ensemble sur une tâche complexe. Par exemple, un serveur web peut créer des processus enfants pour gérer les requêtes des clients, chaque processus enfant traitant une requête spécifique en parallèle avec les autres processus enfants.

Les processus enfants se terminent généralement lorsque leur tâche est terminée ou lorsqu'ils rencontrent une erreur. Le processus parent doit attendre la fin de l'exécution du processus enfant et récupérer son statut de sortie pour éviter que le processus enfant ne devienne un "processus zombie". Un processus zombie est un processus dont l'exécution est terminée, mais qui n'a pas encore été nettoyé par le système d'exploitation parce que son processus parent n'a pas récupéré son statut de sortie.

En résumé, les processus enfants sont des processus créés et gérés par un processus parent. Ils héritent de certaines propriétés du processus parent, mais fonctionnent de manière indépendante, avec leur propre espace mémoire, contexte d'exécution et état. Les processus enfants sont utilisés pour exécuter des tâches parallèles et pour faciliter la répartition du travail entre plusieurs processus.

2.8. Qu'est-ce qu'un signal UNIX ?

Un signal UNIX est un mécanisme de communication utilisé dans les systèmes d'exploitation de type Unix pour informer un processus qu'un événement spécifique s'est produit. Les signaux sont utilisés pour transmettre des informations entre les processus ou entre le système d'exploitation et un processus. Ils permettent aux processus de réagir à des situations spécifiques, telles que l'interruption d'une opération en cours, une erreur fatale ou la fin de l'exécution d'un processus enfant.

Les signaux sont identifiés par des numéros entiers et ont généralement des noms associés. Par exemple, le signal "SIGINT" (signal numéro 2) est utilisé pour interrompre un processus en cours d'exécution, tandis que le signal "SIGTERM" (signal numéro 15) est utilisé pour demander poliment à un processus de se terminer.

Lorsqu'un signal est envoyé à un processus, le système d'exploitation détermine la manière dont le processus doit réagir à ce signal. Un processus peut choisir de :

Ignorer le signal : Le processus continue son exécution normale sans tenir compte du signal. Cependant, certains signaux, comme "SIGKILL" (signal numéro 9) et "SIGSTOP" (signal numéro 19), ne peuvent pas être ignorés.

Attraper le signal : Le processus peut définir une fonction de rappel, appelée "gestionnaire de signal", qui sera exécutée lorsque le signal est reçu. Le gestionnaire de signal peut effectuer des opérations spécifiques en réponse au signal, comme effectuer un nettoyage avant de terminer le processus.

Utiliser l'action par défaut : Si le processus n'a pas défini de gestionnaire de signal pour un signal spécifique, l'action par défaut du signal sera exécutée. Par exemple, l'action par défaut pour le signal "SIGTERM" est de terminer le processus, tandis que l'action par défaut pour le signal "SIGCONT" (signal numéro 18) est de reprendre l'exécution d'un processus suspendu.

Les signaux UNIX sont un moyen simple et efficace de communiquer entre les processus et de gérer les événements asynchrones. Ils sont largement utilisés dans les systèmes d'exploitation Unix et Unix-like pour la gestion des processus, le contrôle des tâches et la gestion des erreurs.

2.9. Qu'est-ce qu'un file descriptor ?

Un descripteur de fichier (file descriptor, en anglais) est un identifiant numérique utilisé par les systèmes d'exploitation pour représenter et gérer l'accès aux fichiers, aux sockets réseau et à d'autres ressources. Les descripteurs de fichiers sont utilisés pour interagir avec ces ressources à travers les appels système et les fonctions de bibliothèque standard.

Lorsqu'un processus ouvre un fichier, crée une socket ou accède à une autre ressource, le système d'exploitation lui attribue un descripteur de fichier unique. Ce descripteur de fichier est ensuite utilisé par le processus pour lire, écrire, modifier ou fermer la ressource.

Dans les systèmes de type Unix, les descripteurs de fichiers sont généralement représentés par des entiers non négatifs. Par convention, trois descripteurs de fichiers sont déjà ouverts pour chaque processus lors de son démarrage :

- Descripteur de fichier **0 (STDIN)** : Il représente l'entrée standard, généralement utilisée pour lire les données fournies par l'utilisateur ou par un autre processus.

- Descripteur de fichier **1 (STDOUT)** : Il représente la sortie standard, utilisée pour écrire des données à destination de l'utilisateur ou d'un autre processus.

- Descripteur de fichier **2 (STDERR)** : Il représente la sortie d'erreur standard, utilisée pour écrire des messages d'erreur ou de diagnostic à destination de l'utilisateur ou d'un autre processus.

Les descripteurs de fichiers permettent aux processus de travailler avec plusieurs fichiers et ressources en même temps, en utilisant un système d'indexation simple et efficace. Ils sont également utilisés dans les mécanismes de communication interprocessus (IPC), tels que les pipes et les redirections, pour transmettre des données entre les processus.

En résumé, un descripteur de fichier est un identifiant numérique utilisé par les systèmes d'exploitation pour représenter et gérer l'accès aux fichiers, aux sockets réseau et à d'autres ressources. Les processus utilisent des descripteurs de fichiers pour interagir avec ces ressources et pour communiquer entre eux.

2.10. Streams vs. File Descriptors

Un stream est une abstraction de haut niveau pour représenter un flux séquentiel de données, utilisé principalement pour la lecture, l'écriture et le traitement de données dans les applications. Les streams sont généralement associés aux opérations d'entrée/sortie (I/O) sur des fichiers, des sockets réseau, des communications inter-processus ou d'autres sources de données. Ils sont conçus pour simplifier et uniformiser la manipulation de données en séquence, quelle que soit la source ou la destination des données.

Dans les langages de programmation et les bibliothèques standard, les streams sont souvent représentés par des objets ou des structures de données qui fournissent des méthodes pour lire, écrire, ouvrir, fermer et manipuler les données en séquence. Par exemple, en C et en C++, les bibliothèques standard fournissent des fonctions de manipulation de fichiers basées sur des streams, telles que fopen, fread, fwrite et fclose.

La différence principale entre un stream et un descripteur de fichier réside dans le niveau d'abstraction :

1. Un descripteur de fichier est un identifiant numérique de bas niveau utilisé par les systèmes d'exploitation pour représenter et gérer l'accès aux fichiers, aux sockets réseau et à d'autres ressources. Les descripteurs de fichiers sont utilisés dans les appels système et les fonctions de bas niveau pour manipuler directement les ressources.

2. Un stream est une abstraction de haut niveau pour représenter un flux de données, utilisée dans les langages de programmation et les bibliothèques standard pour simplifier et uniformiser la manipulation de données en séquence. Les streams sont généralement basés sur des descripteurs de fichiers, mais fournissent une interface plus simple et plus conviviale pour les développeurs.

En résumé, les streams et les descripteurs de fichiers sont tous deux utilisés pour gérer les opérations d'entrée/sortie (I/O) dans les systèmes informatiques, mais ils se situent à des niveaux d'abstraction différents. Les descripteurs de fichiers sont des identifiants numériques de bas niveau utilisés par les systèmes d'exploitation, tandis que les streams sont des abstractions de haut niveau fournies par les langages de programmation et les bibliothèques standard pour faciliter la manipulation de flux de données. Les développeurs travaillent généralement avec des streams lorsqu'ils écrivent du code, car ils offrent une interface plus simple et plus intuitive pour la gestion des opérations d'entrée/sortie.

3. Command Line Interface

3.1. Qu’est-ce qu’un shell ?

Un shell est un programme informatique qui fournit une interface utilisateur pour interagir avec un système d'exploitation (OS). Le shell permet aux utilisateurs d'exécuter des commandes, de gérer des fichiers, de lancer des applications et d'effectuer d'autres tâches essentielles sur un ordinateur. Il existe deux types de shells : les shells graphiques (GUI) et les shells en ligne de commande (CLI).

Les shells graphiques, également appelés environnements de bureau, fournissent une interface utilisateur graphique avec des fenêtres, des icônes et des menus pour interagir avec le système d'exploitation. Les exemples courants de shells graphiques incluent Windows Explorer pour Windows, Finder pour macOS et GNOME ou KDE pour les systèmes d'exploitation Linux.

Les shells en ligne de commande, en revanche, sont des interfaces textuelles où les utilisateurs saisissent des commandes pour interagir avec le système d'exploitation. Ces shells sont souvent utilisés par les administrateurs système, les développeurs et les utilisateurs avancés pour des tâches plus complexes ou pour automatiser des processus. Les exemples courants de shells en ligne de commande incluent Bash, Zsh et PowerShell.

En résumé, un shell est un outil essentiel qui permet aux utilisateurs de communiquer avec un système d'exploitation et d'exécuter des tâches variées en utilisant soit une interface graphique, soit une interface en ligne de commande.

3.2. Bash (Bourne-Again SHell), c’est quoi ?

Bash (Bourne-Again SHell) est un shell en ligne de commande (CLI) pour les systèmes d'exploitation de type Unix, comme Linux et macOS. Il a été créé par Brian Fox et est sorti en 1989 en tant que successeur du shell Bourne (sh). Bash est l'un des shells les plus populaires et largement utilisés, en particulier dans les environnements Linux.

Bash permet aux utilisateurs d'interagir avec le système d'exploitation en saisissant des commandes textuelles, de gérer des fichiers et des répertoires, d'exécuter des scripts shell et d'automatiser des tâches. Bash est également un langage de script qui permet aux utilisateurs de créer des scripts pour effectuer des opérations plus complexes ou répétitives. Les scripts Bash peuvent être utilisés pour l'administration système, le développement de logiciels et d'autres tâches informatiques.

En résumé, Bash est un shell en ligne de commande et un langage de script pour les systèmes d'exploitation de type Unix. Il est largement utilisé pour l'administration système, l'exécution de commandes et la création de scripts pour automatiser des tâches et des processus.

3.3. Implémentation de Bash

1. **Input** : L'utilisateur saisit une commande dans l'interface en ligne de commande de Bash. Cette commande est lue et stockée en mémoire sous forme de chaîne de caractères.

2. **Lexer (Analyse lexicale)** : Le lexer divise la chaîne de caractères en mots et opérateurs de contrôle appelés tokens. Il détecte les caractères spéciaux, les espaces, les guillemets, les caractères d'échappement et autres éléments syntaxiques pour décomposer la commande en unités logiques.

3. **Parser (Analyse syntaxique)** : Le parser analyse la structure des tokens pour vérifier qu'ils sont correctement organisés en termes de syntaxe. Il construit une structure de

données appelée Abstract Syntax Tree (AST) pour représenter la commande et ses composants. L'AST est une représentation hiérarchique de la commande et de ses sous-expressions.

4. **Expansion** : Bash effectue plusieurs types d'expansions pour traiter et remplacer les expressions par leurs valeurs appropriées :

    - Expansion des accolades (Brace Expansion)
    - Expansion des tilde (~) pour les répertoires personnels
    - Expansion des paramètres pour les variables
    - Expansion des jokers (Wildcard Expansion) pour les caractères génériques
    - Expansion arithmétique pour les calculs mathématiques
    - Substitution de commandes pour exécuter une commande et utiliser sa sortie
    - Expansion des processus pour exécuter des commandes en parallèle

5. **Gestion du PATH** : Bash utilise la variable d'environnement PATH pour rechercher les commandes externes. Le PATH est une liste de répertoires séparés par des deux-points (":"), où Bash cherche les exécutables correspondant à la commande saisie.

6. **Hash table** : Bash maintient une table de hachage qui stocke les chemins des exécutables déjà trouvés pour accélérer la recherche des commandes externes. Cela permet de réduire le temps nécessaire pour rechercher à nouveau dans le PATH pour les commandes fréquemment utilisées.

7. **Exécution** : Bash exécute la commande, en fonction de son type :

    - Commandes internes : intégrées au shell, elles sont directement exécutées par Bash.
    - Commandes externes : Bash lance un nouveau processus pour exécuter l'exécutable trouvé dans le PATH.
    - Fonctions : définies par l'utilisateur, elles sont exécutées dans le contexte du shell courant.

8. **Redirections et pipes** : Bash gère les redirections et les pipes pour modifier les entrées et les sorties des commandes :

    - Redirections de l'entrée standard (stdin) pour lire les données depuis un fichier
    - Redirections de la sortie standard (stdout) et de la sortie d'erreur (stderr) pour écrire les données dans un fichier
    - Pipes (|) pour connecter la sortie d'une commande à l'entrée d'une autre

9. **Contrôle de flux** : Bash prend en charge les structures de contrôle de flux telles que les boucles (for, while), les tests conditionnels (if, elif, else), et les opérateurs de contrôle (&&, ||) pour permettre l'exécution séquentielle ou conditionnelle des commandes. Ces

structures de contrôle sont utilisées pour répéter des actions, exécuter des commandes en fonction de conditions spécifiques, ou combiner des commandes de manière logique.

10. **Gestion des processus** : Bash gère les processus en arrière-plan (background) et en avant-plan (foreground), permettant ainsi l'exécution simultanée de plusieurs commandes. L'utilisateur peut également suspendre des processus en cours, les reprendre en arrière-plan ou en avant-plan, et envoyer des signaux aux processus pour les contrôler (par exemple, SIGINT pour interrompre, SIGTERM pour terminer).

11. **Gestion des variables d'environnement** : Bash permet la définition, la modification et l'utilisation de variables d'environnement pour stocker des informations globales et spécifiques au processus. Les variables d'environnement sont utilisées pour configurer le comportement du shell et des programmes exécutés à partir du shell.

12. **Gestion des alias** : Bash permet la création d'alias, qui sont des raccourcis pour les commandes, permettant à l'utilisateur de définir des abréviations ou des chaînes de commandes personnalisées pour faciliter l'utilisation du shell.

13. **Historique des commandes** : Bash conserve un historique des commandes saisies par l'utilisateur, permettant ainsi de naviguer facilement dans les commandes précédemment utilisées et de les réexécuter si nécessaire. L'historique des commandes peut également être recherché, filtré et modifié.

14. **Complétion automatique** : Bash offre une fonction de complétion automatique pour les noms de fichiers, les répertoires, les variables et les commandes. L'utilisateur peut appuyer sur la touche Tab pour compléter automatiquement une saisie partielle, ce qui facilite et accélère la navigation et l'exécution de commandes.

15. **Scripts** : Bash permet l'exécution de scripts shell, qui sont des séquences de commandes sauvegardées dans un fichier. Les scripts peuvent être utilisés pour automatiser des tâches répétitives, exécuter des séquences de commandes complexes, ou créer des outils personnalisés.

En résumé, l'implémentation de Bash implique un certain nombre de composants et d'étapes pour lire, analyser, traiter et exécuter les commandes entrées par l'utilisateur. Ces composants incluent l'analyse lexicale et syntaxique, l'expansion, la gestion du PATH, l'exécution des commandes, la gestion des redirections et pipes, le contrôle de flux, la gestion des processus, la gestion des variables d'environnement, des alias, l'historique des commandes, la complétion automatique et l'exécution des scripts.

3.4. Commandes, exécutables et builtins

Dans le contexte d'un shell comme Bash, les termes "commande", "exécutable" et "builtin" ont des significations spécifiques :

1. Commande : Une commande est une instruction donnée par l'utilisateur au shell pour effectuer une certaine tâche. Elle peut être une simple instruction pour afficher un fichier, créer un répertoire, gérer des processus, ou exécuter un programme. Les commandes sont généralement composées du nom de la commande suivi d'options, d'arguments, et d'autres éléments de syntaxe pour décrire l'action souhaitée. Les commandes peuvent être de plusieurs types, y compris les commandes internes (builtins), les commandes externes (exécutables) et les fonctions définies par l'utilisateur.

2. Exécutable : Un exécutable est un fichier contenant un programme qui peut être exécuté par le système d'exploitation. Les exécutables sont des commandes externes qui ne sont pas intégrées au shell. Lorsqu'une commande externe est entrée, le shell recherche le fichier exécutable correspondant dans les répertoires spécifiés par la variable d'environnement PATH. Une fois que le fichier exécutable est trouvé, Bash lance un nouveau processus pour exécuter ce programme. Les exécutables peuvent être des fichiers binaires compilés ou des scripts interprétés par d'autres programmes (comme les scripts Python ou Perl).

3. Builtin (commande interne) : Les commandes internes sont des fonctionnalités intégrées au shell lui-même et ne nécessitent pas de lancer un processus externe pour être exécutées. Ces commandes sont généralement des opérations de base qui sont fréquemment utilisées ou qui interagissent directement avec le shell. Par exemple, cd (pour changer de répertoire), echo (pour afficher du texte), set (pour définir des variables) et exit (pour quitter le shell) sont des commandes internes. Les commandes internes sont exécutées directement par Bash et sont généralement plus rapides que les commandes externes, car elles ne nécessitent pas de création de nouveaux processus.

En résumé, une commande est une instruction donnée par l'utilisateur pour effectuer une action, un exécutable est un fichier contenant un programme qui peut être lancé en tant que commande externe, et un builtin est une commande interne intégrée au shell lui-même.

3.5. Environnement d'un shell

L'environnement d'un shell fait référence à un ensemble de variables et de configurations qui déterminent le comportement du shell et des programmes lancés à partir du shell. L'environnement inclut des variables d'environnement, des alias, des fonctions et d'autres paramètres qui sont utilisés pour personnaliser l'expérience utilisateur, configurer les chemins d'accès aux exécutables et contrôler divers aspects des programmes exécutés.

Les variables d'environnement sont des paires clé-valeur qui stockent des informations globales accessibles à la fois par le shell et les processus lancés à partir du shell. Les variables d'environnement sont utilisées pour transmettre des informations telles que les préférences de l'utilisateur, les paramètres de configuration et les chemins d'accès aux ressources du système. Certaines variables d'environnement courantes incluent :

- **PATH** : une liste de répertoires séparés par des deux-points (":"), où le shell recherche les exécutables des commandes externes.
- **HOME** : le chemin d'accès au répertoire personnel de l'utilisateur.
- **USER** : le nom d'utilisateur actuel.
- **LANG** : la langue et les paramètres régionaux pour le système.
- **TERM** : le type de terminal utilisé pour l'affichage du shell.

Les alias sont des raccourcis pour les commandes, permettant à l'utilisateur de définir des abréviations ou des chaînes de commandes personnalisées pour faciliter l'utilisation du shell.

Les fonctions sont des blocs de code définis par l'utilisateur qui peuvent être appelés comme des commandes. Elles sont utilisées pour encapsuler des séquences de commandes réutilisables, simplifiant ainsi les tâches complexes ou répétitives.

L'environnement d'un shell peut être hérité par les processus lancés à partir du shell, ce qui permet aux processus enfants d'accéder aux variables d'environnement et autres configurations définies dans l'environnement du shell.

En résumé, l'environnement d'un shell est constitué d'un ensemble de variables et de configurations qui déterminent le comportement du shell et des programmes lancés à partir de celui-ci. Il inclut des variables d'environnement, des alias et des fonctions, qui sont utilisés pour personnaliser l'expérience utilisateur et contrôler les aspects des programmes exécutés.

3.6. Qu'est-ce qu'un chemin relatif et un chemin absolu (Unix) ?

Les chemins sont utilisés pour localiser des fichiers et des dossiers dans le système de fichiers. Ils sont essentiels pour naviguer et manipuler les fichiers. Il existe deux types de chemins : les chemins relatifs et les chemins absolus.

1. Chemin relatif :
   Un chemin relatif est un chemin qui commence par rapport au répertoire de travail actuel. Il ne commence pas par la racine du système de fichiers. Par exemple, si vous êtes dans le répertoire **`/home/user/documents`**, et que vous souhaitez accéder au dossier **`images`** à l'intérieur de **`documents`**, le chemin relatif serait simplement **`images`**. Un autre exemple de chemin relatif est **`../music`**, qui représente le dossier **`music`** situé dans le répertoire parent du répertoire de travail actuel.

2. Chemin absolu :
   Un chemin absolu est un chemin qui commence par la racine du système de fichiers (représentée par `/` dans Unix). Il fournit un emplacement précis et unique pour un fichier ou un dossier dans le système. Par exemple, `**/home/user/documents/images**` est un chemin absolu qui pointe vers le dossier images indépendamment du répertoire de travail actuel.

En résumé, un chemin relatif commence par rapport au répertoire de travail actuel, tandis qu'un chemin absolu commence à partir de la racine du système de fichiers et spécifie un emplacement unique et précis. Les deux types de chemins sont utilisés pour accéder et manipuler les fichiers et les dossiers dans un système Unix.

3.7. Pourquoi exécute-t-on une commande dans un processus enfant ?

L'exécution d'une commande dans un processus enfant est une pratique courante en programmation pour plusieurs raisons. Voici quelques-unes des raisons principales :

1. Isolation : Les processus sont indépendants les uns des autres et ont leur propre espace mémoire. Lorsqu'une commande est exécutée dans un processus enfant, cela évite que les ressources du processus parent ne soient affectées par les erreurs ou les problèmes de performance qui pourraient survenir dans le processus enfant. Si le processus enfant rencontre une erreur fatale, le processus parent reste généralement intact.

2. Parallélisme : Exécuter une commande dans un processus enfant permet de tirer parti du multitâche, ce qui améliore les performances globales de l'application. Les processus parent et enfant peuvent s'exécuter simultanément et utiliser plusieurs cœurs de processeur, ce qui permet un traitement plus rapide et une meilleure utilisation des ressources du système.

3. Contrôle : En utilisant des processus enfants, il est plus facile de contrôler et de gérer les différentes tâches. Par exemple, le processus parent peut surveiller l'état et les ressources du processus enfant, envoyer des signaux pour arrêter ou redémarrer le processus enfant si nécessaire, et récupérer les résultats du processus enfant lorsqu'il a terminé.

4. Modularité : Les processus enfant permettent de découpler les fonctionnalités et de créer des applications modulaires. Les développeurs peuvent créer des composants indépendants qui peuvent être exécutés et testés séparément, facilitant ainsi la maintenance et l'évolutivité du code.

Dans les systèmes d'exploitation Unix et Linux, on utilise souvent la méthode "fork-exec" pour créer un processus enfant. La fonction "fork" crée une copie du processus parent, et la fonction "exec" remplace l'image du processus enfant par le programme à exécuter.

Dans le cadre de minishell, nous pouvons souvent nous retrouver à devoir exécuter plusieurs commandes en parallèle (cf. pipeline), garder et rediriger parfois les résultats entre chaque commande. Dans ce contexte, nous n'avons d'autres choix que d'utiliser des processus enfants entre chaque commande, et de communiquer entre les processus enfants grâce aux pipes.

3.8. Les redirections c'est quoi ?

Dans le contexte de bash (un interpréteur de commandes Unix), les redirections permettent de modifier la manière dont les entrées et les sorties des commandes sont gérées. Les entrées et les sorties sont normalement traitées via des fichiers spéciaux appelés descripteurs de fichiers. Par défaut, il y a trois descripteurs de fichiers ouverts pour chaque processus :

- 0 : Entrée standard (stdin)
- 1 : Sortie standard (stdout)
- 2 : Erreur standard (stderr)

Les redirections vous permettent de modifier ces entrées et sorties, par exemple, pour enregistrer les résultats d'une commande dans un fichier ou lire les données depuis un fichier.

Voici quelques types de redirections possibles :

1. Redirection de sortie (>): Cette redirection envoie la sortie standard d'une commande vers un fichier. Si le fichier existe déjà, son contenu sera écrasé.
   Exemple : **`ls > fichier.txt`**

2. Redirection d'entrée (<): Cette redirection lit les données depuis un fichier et les utilise comme entrée standard pour une commande.
   Exemple : **`sort < fichier.txt`**

3. Redirection de sortie en mode append (>>): Semblable à la redirection de sortie, mais au lieu d'écraser le contenu du fichier, elle ajoute la sortie à la fin du fichier.
   Exemple : **`ls >> fichier.txt`**

4. Redirection d'erreurs (2>): Cette redirection envoie les messages d'erreur (stderr) vers un fichier ou un autre descripteur de fichier.
   Exemple : **`ls non_existant 2> erreurs.txt`**

5. Heredoc (<<): C'est une méthode pour fournir un document en entrée standard (stdin) à une commande en utilisant un délimiteur spécifique.

Exemple :

`cat << EOF
Texte à inclure
dans le fichier.
EOF`

6. Redirection combinée (>&): Cette redirection permet de combiner stdout et stderr pour les diriger vers un fichier ou un autre descripteur de fichier.
   Exemple : **`ls non_existant > resultat.txt 2>&1`**

Ces redirections peuvent être combinées pour effectuer des opérations plus complexes, telles que rediriger à la fois la sortie standard et les erreurs vers un fichier.