Document Compagnon

Le recueil des 4 séquences

Session 3

Version 3.4

Un MOOC de l'université de La Réunion

UNIVERSITÉ DE LA RÉUNION

Licence Creative Commons CC BY-NC-SA 4.0 International

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International (CC BY-NC-SA 4.0)

Avertissement Ce résumé n'indique que certaines des dispositions clé de la licence. Ce n'est pas une licence, il n'a pas de valeur juridique. Vous devez lire attentivement tous les termes et conditions de la licence avant d'utiliser le matériel licencié.
Creative Commons n'est pas un cabinet d'avocat et n'est pas un service de conseil juridique. Distribuer, afficher et faire un lien vers le résumé ou la licence ne constitue pas une relation client-avocat ou tout autre type de relation entre vous et Creative Commons.

Clause : C'est un résumé (et non pas un substitut) de la licence.
`https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode` **Vous êtes autorisé à :**

- **Partager** - copier, distribuer et communiquer le matériel par tous moyens et sous tous formats,
- **Adapter** - remixer, transformer et créer à partir du matériel,
- pour une utilisation non commerciale.

L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :

Attribution (BY) - Vous devez créditer l'oeuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'oeuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son Oeuvre.

Pas d'Utilisation Commerciale (NC) - Vous n'êtes pas autorisé à faire un usage commercial de cette Oeuvre, tout ou partie du matériel la composant.

Partage dans les mêmes Conditions (SA) - Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'Oeuvre originale, vous devez diffuser l'Oeuvre modifiée dans les même conditions, c'est à dire avec la même licence avec laquelle l'Oeuvre originale a été diffusée.

Pas de restrictions complémentaires - Vous n'êtes pas autorisé à appliquer des conditions légales ou des **mesures techniques** qui restreindraient légalement autrui à utiliser l'Oeuvre dans les conditions décrites par la licence.

Notes : Vous n'êtes pas dans l'obligation de respecter la licence pour les éléments ou matériel appartenant au domaine public ou dans le cas où l'utilisation que vous souhaitez faire est couverte par une **exception**.

Aucune garantie n'est donnée. Il se peut que la licence ne vous donne pas toutes les permissions nécessaires pour votre utilisation. Par exemple, certains droits comme les **droits moraux, le droit des données personnelles et le droit à l'image** sont susceptibles de limiter votre utilisation.

Les informations détaillées sont disponibles aux URL suivantes :

- `http://creativecommons.org/licenses/by-sa/4.0/deed.fr`
- `http://fr.wikipedia.org/wiki/Creative_Commons`

Auteurs de cette session

Pascal ANELLI est professeur à l'Université de la Réunion. Il enseigne l'informatique depuis plus de 20 ans. Il utilise les outils d'Unix et en particulier le Bash dans le cadre de ses travaux de recherche. Il a formé des centaines d'étudiants à ce langage.

Régis GIRARD est maître de conférences à l'Université de La Réunion. Il a enseigné Unix et le shell pendant plusieurs années en Licence d'Informatique.

Contributeurs

Isabelle POIRIER est professeur agrégé de mathématiques et enseigne celles-ci depuis plus de 15 ans en établissement secondaire dans le sud de la France. Autodidacte en programmation (en particulier grâce à sa participation à plusieurs MOOC), elle met à profit son sens de la pédagogie et sa propre expérience pour venir en aide aux apprenants sur le forum et améliorer l'approche didactique proposée au bénéfice d'un meilleur apprentissage.

Auteurs des sessions précédentes

Xavier NICOLAY est ingénieur de recherche à l'Université de La Réunion. Il a été ingénieur systèmes & réseaux pendant 20 ans puis Directeur Informatique.

Denis PAYET est maître de conférences à l'Université de La Réunion, il enseigne dans le domaine du génie logiciel et de la programmation en intervenant sur les différentes formations de l'université : DUT, Licence, Master et dans le cycle Ingénieur.

Tahiry RAZAFINDRALAMBO est maître de conférences à l'Université de La Réunion, il enseigne l'informatique et notamment l'automatisation des tâches d'administration sous Linux et Unix. Il utilise le Bash dans la majorité de ses travaux et l'enseigne aux étudiants.

Pierre-Ugo TOURNOUX est maître de conférences à l'Université de La Réunion, il enseigne l'informatique et notamment l'administration système et serveur. Le Bash est la pierre angulaire de la plupart de ses enseignements.

Table des activités

Séquence 0

Bienvenue

Activité 0.1

Un cours orienté pratique

1 Motivations à l'utilisation du shell

Le shell Bash est désormais omniprésent car il a été intégré à Windows 10 depuis une mise à jour récente. Ainsi, dorénavant, les 3 grands systèmes d'exploitation pour ordinateur (Windows, MacOS et Linux) partagent cet outil puissant et incontournable pour tout informaticien, que ce soit à l'échelle du poste de travail, d'un serveur et même d'un smartphone ou d'une tablette. L'essor des terminaux Android permet en effet également d'intégrer la dimension mobile au spectre d'utilisation du shell. On peut aussi noter l'engouement pour les nano-ordinateurs comme les Raspberry PI et plus généralement pour l'instrumentation connectée (ou Internet des objets) qui fonctionnent presque exclusivement sous une distribution Linux. L'utilisation du shell est donc largement transversale et sa connaissance est un passage obligé pour tous les informaticiens.

2 Pourquoi connaître le shell ?

La science informatique revêt plusieurs aspects, un des plus fondamentaux porte sur le langage pour écrire des programmes. Dans la multitude des langages de programmation, le langage de programmation shell occupe une place à part. Et pour bien marquer cette différence, on parle de script et non de programme. En effet, en plus d'être un langage de programmation, le shell est aussi un interpréteur de commandes accessible depuis la console (aussi appelé terminal). Le shell offre alors une interface homme-machine pour décrire, sous la forme de lignes de commande, les actions à effectuer au sein du système d'exploitation. Cette interface textuelle nécessite de la pratique pour être maîtrisée et elle est souvent ressentie comme complexe à apprendre. Par la flexibilité et la richesse qu'elle propose, elle est indispensable pour celui qui souhaite administrer et configurer un ordinateur personnel ou un serveur sous Linux.

La programmation de script shell répond à la philosophie Unix classique consistant à diviser les tâches complexes en sous-tâches plus simples, puis à chaîner des composants et des utilitaires. Ces utilitaires sous forme de commandes sont comme une boite à outils disponible dans les systèmes de type Unix. Beaucoup considèrent que c'est la meilleure approche, ou tout au moins plus agréable pour la résolution d'un problème que d'avoir recours à écrire un programme complet avec un langage puissant et généraliste. Cela impose cependant une modification du mode de réflexion, qui doit être adapté aux utilitaires disponibles.

3 Objectif du cours

En utilisant le shell Bash, vous verrez comment la console est complémentaire de l'interface graphique et par la richesse du shell Bash, vous découvrirez comment améliorer votre expérience utilisateur sur votre ordinateur. Ce cours a pour objectif de vous rendre capable d'utiliser une console pour les opérations courantes sur votre ordinateur. L'expression de vos actions s'effectue au moyen d'un langage de commande. Nous vous proposons de vous familiariser avec le shell Bash. Ce shell est la version la plus courante. Un langage quel qu'il soit s'apprend en le pratiquant - c'est pourquoi nous adoptons une approche orientée vers la pratique.

À l'issue de ce cours vous serez capable de lire, d'écrire et de comprendre des commandes dans le contexte du shell Bash. Mais au delà de l'interaction classique avec le shell Bash, ce cours vous propose d'acquérir les compétences nécessaires pour exploiter toute sa puissance dans des scripts. Vous aurez acquis les connaissances suffisantes pour pouvoir interagir avec votre ordinateur sans passer par une interface graphique. De plus, vous maîtriserez les fondamentaux d'un langage de script pour en apprendre d'autres facilement par vous-même.

4 Contenu du cours

Langages interprétés

Dans ces langages, le code informatique appelé code source, que vous écrivez, est traité par un logiciel : l'interpréteur. Le rôle de ce dernier va consister à lire et à exécuter les lignes du programme une par une. À chaque lancement du programme à interpréter, l'interpréteur effectue cette traduction du code source en un code exécutable par l'ordinateur.

Une connaissance pratique de la programmation de scripts shell est indispensable à quiconque souhaitant administrer un système informatique. Les scripts shell peuvent servir à mettre en place des procédures pour le système mais également constituer des programmes utilitaires. Le script shell va vous permettre d'écrire des prototypes d'application, de programmer des tâches courantes ou d'automatiser des enchaînements de programmes.

L'apprentissage des scripts n'est pas compliqué puisque les scripts shell peuvent se construire progressivement par petit morceaux par le principe même des langages interprétés. La syntaxe d'un script est simple même si elle peut paraître austère. Elle est identique à celle utilisée en ligne de commande. Cette syntaxe ne demande que de connaître quelques règles et des opérateurs spécifiques. La plupart des scripts courts fonctionnent la première fois, et la mise au point des scripts les plus importants reste simple.

Le cours couvre tous les aspects du shell et de son langage de programmation. Vous commencerez par étudier pendant les 3 premières séquences :

- les éléments d'un système d'exploitation ;
- la syntaxe spécifique du shell ;
- les utilitaires du système d'exploitation.

La dernière séquence sera consacrée à la présentation des structures de programmation pour la réalisation de scripts shell.

5 Organisation

L'approche pédagogique de ce cours se veut orientée sur l'utilisation courante du shell et sur la présentation des principales commandes d'un système de type Unix. Il peut être tentant de présenter l'ensemble des commandes et des constructions disponibles par un shell. Vu le nombre, cela peut vite

devenir un catalogue. Nous avons préféré nous concentrer sur l'essentiel, à savoir sur la présentation des concepts sous-tendant les commandes présentées. Nous avons recours à de nombreux exemples pour illustrer les éléments du langage.

Notre démarche de présentation est la suivante :

- Après un exposé des concepts illustrés par des commandes à l'aide de la vidéo, un support de cours reprend plus en détail les éléments de langage exposés par la vidéo. Ce support de cours que nous appellerons *document compagnon* constitue un complément indispensable à la vidéo.
- Une console identique à celle utilisée par l'enseignant est proposée pour refaire les exemples.
- Des exercices sous la forme de challenges sont à réaliser afin de pratiquer les notions abordées et de les acquérir immédiatement. Ces travaux pratiques entrent dans le cadre de l'évaluation qui vous permettra d'obtenir une attestation de succès. Vous disposez d'une console Linux dans votre navigateur pour vous exercer et réaliser ces challenges. Cet environnement Linux simplifié et commun à tous les apprenants vous permet de faire vos premières manipulations sans risque. À la fin du cours, vous pourrez vous lancer sans crainte dans un système complet.
- Le document compagnon propose des exercices supplémentaires avec des corrections. Ils sont là pour vous entraîner aux challenges.
- Des quiz sont à remplir pour valider les compétences que vous acquerrez au fur et à mesure de votre progression. Ils font aussi partie de l'évaluation en vue de l'attestation de réussite.
- Un forum de discussion sera ouvert avec un fil de discussion relatif à chaque thème abordé. N'hésitez pas à aller sur le forum pour poser des questions, faire des remarques, obtenir des informations ou des éclaircissements supplémentaires sur le cours. La communauté du MOOC est là pour vous aider.

Ce cours se déroule sur un peu plus de 3 mois et comporte, en plus de la séquence de bienvenue, 4 séquences thématiques composées chacune d'une introduction, de 5 activités et d'une conclusion. Le contenu de chaque séquence est développé dans un document compagnon disponible sous la forme d'un fascicule.

Tous les contenus sont accessibles dès l'ouverture et le resteront après la fermeture. Vous disposez de plus de 3 mois pour mener à terme ce MOOC en vue de l'attestation. Seules les 6 premières semaines seront accompagnées par l'équipe enseignante au travers des forums. Ces derniers resteront ouverts durant tout le cours.

Par ailleurs, et dans le cadre de l'animation de votre MOOC, un enseignant vous proposera éventuellement une conférence en ligne sur une démonstration. Vous pourrez suivre directement en ligne cette conférence, ou avoir accès à l'enregistrement qui sera accessible à partir d'un lien qui vous sera communiqué. Cette conférence sera annoncée dans l'info-cours.

Enfin, les apprenants ayant obtenu un score global supérieur à 70% sur les évaluations obtiendront une attestation de suivi avec succès attribuée par FUN.

5.1 Comment passer les évaluations ?

Le quiz de chaque activité porte exclusivement sur la vidéo associée à l'activité. Il sert à l'apprenant à valider sa compréhension du thème présenté à partir de la vidéo. La vidéo donne une vision générale. Il est souvent nécessaire d'avoir consulté le document compagnon pour avoir une compréhension plus complète du thème présenté. Et de manière générale, les challenges pratiques s'appuient sur des compétences acquises avec la lecture du document compagnon.

C'est après avoir effectué le quiz de chaque vidéo, consulté le document compagnon et réalisé les challenges pratiques de la séquence que vous pouvez passer le devoir de fin de séquence. Ainsi votre progression sera évaluée à l'aide des 3 types de travaux :

- Les quiz portant sur la vidéo d'une activité, valant 1% chacun. Il y en a 20, donc les quiz comptent pour 20% de la note finale.

— Les challenges pratiques d'une activité, valant 3% chacun. Il y a en tout 20 challenges répartis en 4 séquences. Ainsi l'ensemble des challenges d'une séquence vaut pour 15% de la note. Et tous les challenges comptent pour 60% de la note finale.
— Les devoirs de fin de séquence, valant 5% chacun. Il y en a 4, donc les devoirs comptent pour 20% de la note finale.

Les questions des quiz et des challenges tolèrent 3 tentatives de réponse, les questions des devoirs de fin de séquence 2 tentatives. Après avoir utilisé toutes les tentatives, la bonne réponse est donnée avec les explications.

6 À qui s'adresse ce cours ?

Vous êtes étudiant en informatique, autodidacte en informatique, ou plus généralement une personne s'initiant ou se perfectionnant à un système d'exploitation de type Unix (dont le Bash permet le pilotage). Ce cours s'adresse à vous tous, informaticiens en devenir ou spécialistes confirmés. Vous avez en commun de vouloir découvrir et maîtriser le langage de l'interaction avec un système de type Unix.

7 Pré-requis

Le pré-requis pour suivre ce cours est de posséder des notions simples de programmation. Les structures et les principes de la programmation doivent être connus afin de comprendre comment ils sont décrits par le shell. Ce cours s'adresse donc à des apprenants ayant des connaissances minimales de programmation.

Cependant, nous sommes convaincus qu'une forte motivation et un travail sérieux peuvent remplacer ces pré-requis.

La section « Testez vos connaissances » vous permet de vérifier si votre niveau de connaissance est suffisant pour commencer ce cours dans les meilleures conditions.

Activité 0.2

Comment le cours est-il structuré ?

Le MOOC Maîtriser le shell Bash vous donne les bases pour interagir avec un ordinateur à l'aide d'un langage de commande. Au moyen des vidéos pédagogiques et des travaux pratiques, vous allez acquérir les compétences nécessaires pour exprimer vos actions par un langage adapté pour le dialogue avec l'ordinateur et au-delà, pour programmer des scripts pour automatiser les tâches courantes. Vous serez ainsi capable d'exploiter toute la puissance d'expression et la flexibilité du shell.

Ce cours est structuré en 4 séquences thématiques comportant chacune une introduction, 5 activités et une conclusion. Préalablement à ces séquences, le cours commence par une séquence de bienvenue. La structuration de ce cours est la suivante :

1. Séquence de bienvenue présentée par Pascal Anelli & Denis Payet. Cette semaine traite de :
   — Présentation du cours
   — Qu'est-ce qu'un shell ?
   — Les challenges pratiques
   — Pour naviguer sur FUN
   — Faisons connaissance
   — Mieux vous connaître
   — Testez vos connaissances
   — Réussir votre MOOC
2. Séquence 1 "Découvrez votre système d'exploitation" présentée par Pierre-Ugo Tournoux
   — Qu'est ce que la ligne de commande ?
   — Trouver de l'aide
   — Gérer les répertoires et les fichiers
   — Les utilisateurs et leurs droits
   — Traiter un fichier de texte
3. Séquence 2 "Interagissez avec le Bash" présentée par Pascal Anelli & Régis Girard
   — Aide à l'interaction
   — Abréviations pour le nom des fichiers
   — Constructions syntaxiques
   — Contrôler l'exécution des commandes
   — Entrées et sorties des processus
4. Séquence 3 "Maîtrisez votre système d'exploitation par le Bash" présentée par Tahiry Razafindralambo
   — Contrôler son environnement
   — Filtres simples
   — Filtres puissants
   — Effectuer des calculs numériques
   — Archiver et compresser des données

5. Séquence 4 "Automatisez vos travaux" présentée par Denis Payet & Régis Girard
   — Éléments d'un script shell
   — Expressions et conditions
   — Structures conditionnelles
   — Structures itératives
   — Structures de routines

À l'issue de ce cours, le shell Bash va faire partie de votre quotidien.

Activité 0.3

Qu'est ce qu'un shell ?

1 Introduction

1.1 C'est quoi ?

Habituellement, quand vous êtes devant votre ordinateur, vous communiquez avec lui en utilisant des modalités d'interaction graphique. Vous avez en effet à votre disposition une souris avec laquelle vous faites bouger un curseur à l'écran pour sélectionner des items et agir sur eux à l'aide de menus. Vous pouvez agir de cette façon parce que le système d'exploitation (SE ou en anglais OS pour *Operating System*) de votre ordinateur dispose d'une interface graphique qui affiche les éléments de votre ordinateur, et vos différents programmes, dans des fenêtres contenant des boutons et des menus.

Programme application et logiciel

Les termes : programme, application et logiciel sont des synonymes. Mais il existe cependant des petites nuances qui font qu'on est parfois amené à choisir l'un d'eux plutôt qu'un autre : le mot **programme** met l'accent sur les traitements informatiques réalisés, le terme **application** insiste sur le fait que l'utilisateur final est le principal bénéficiaire, alors que **logiciel** est souvent employé pour marquer une distinction par rapport à la couche matérielle.

Tous les systèmes d'exploitation modernes disposent de ce type d'interface. C'est effectivement plus facile pour l'utilisateur humain de visualiser les choses ainsi, et d'indiquer par des gestes au système ce qu'il doit faire. C'est plus facile pour l'humain, mais ce n'est pas aussi facile pour le système lui-même, car le système ne fonctionne pas en se représentant les choses visuellement comme un humain le fait. Aussi, l'interface graphique est une surcouche logicielle qui est chargée de traduire les informations du système sous forme d'images. Cette interface se charge en retour d'interpréter les actions de l'utilisateur effectuées à l'aide de la souris et du clavier en actions à réaliser par le système interne.

L'interface graphique est ainsi une sorte d'interprète entre l'utilisateur et le système d'exploitation. Bien que l'interface graphique facilite les choses, ça ne les rend pas pour autant forcément efficaces. En effet, cet interprète ne laisse à l'utilisateur que les actions que le concepteur de l'interface a bien voulu lui laisser. Le champ des possibles est forcément limité. De plus, pour fonctionner, cet interprète consomme des ressources de l'ordinateur. Et ces ressources ne sont pas toujours disponibles ou accessibles, comme c'est le cas pour les accès distants depuis un autre ordinateur. Alors, quand on maîtrise l'interprétation des informations que l'on recherche, ou l'action que l'on souhaite déclencher, il est souvent plus efficace de se passer de cet intermédiaire et d'opérer plus directement avec le système. La question est donc : ce moyen de dialogue plus direct existe-t-il ? Et la réponse est : oui, c'est **le shell**.

La définition donnée du shell par wikipédia est : « la couche logicielle qui fournit l'interface utilisateur d'un système d'exploitation. C'est un interpréteur de commandes destiné aux systèmes d'exploitation

Unix et de type Unix qui permet d'accéder aux fonctionnalités internes du système d'exploitation. Il se présente sous la forme d'une interface en ligne de commande accessible depuis la console ou un terminal. L'utilisateur lance des commandes sous forme d'une entrée texte exécutée ensuite par le shell. »

1.2 L'origine du shell

Historiquement, les premiers systèmes d'exploitation ne disposaient pas d'interface graphique. D'ailleurs, à cette époque, la souris n'existait même pas. On interagissait donc avec le système essentiellement par échange de texte. L'écran servait à recevoir les informations fournies par le système sous forme de phrase, d'affichage de valeur ou de tableau textuel. Et l'utilisateur utilisait le clavier pour saisir les ordres qu'il voulait soumettre au système. C'est cette interface d'échange textuel qu'on appelle le shell.

Parler au passé du shell ? En fait, non, le shell existe encore. Il règne en maître depuis toujours sur la quasi-totalité des serveurs web que vous visitez, des ordinateurs des systèmes informatiques des entreprises et des administrations, ou encore sur les serveurs qui fournissent les services que vous utilisez quotidiennement sur Internet (Cloud, Twitter, etc.).

1.3 Pourquoi s'intéresser au shell ?

Le shell n'a jamais vraiment disparu, il a simplement été de moins en moins visible par le grand public. Essentiellement parce que pour le grand public l'accent a été mis sur la facilité d'usage plus que sur l'efficacité et la puissance de cet usage. Mais les choses ont déjà commencé à changer, car de plus en plus d'utilisateurs veulent aller plus loin. Ils sont de plus en plus exigeants et de plus en plus compétents, car comme vous, ce sont des passionnés qui cherchent à contrôler toujours mieux leur ordinateur. À tel point que l'on constate aujourd'hui que le shell est en train de faire son grand retour : notamment avec les objets connectés et les Raspberry Pi, mais aussi sur des systèmes d'exploitation propriétaires tels que Windows 10 dont la dernière mise à jour propose un accès plus immédiat au shell.

Pour ces passionnés, et donc pour vous, dans de nombreuses situations les interfaces graphiques vont très vite devenir un véritable frein. Vous aurez donc besoin d'un moyen d'échange plus intime avec le système d'exploitation. Vous aurez besoin du shell. C'est pourquoi nous vous proposons de faire la connaissance de cet outil, et avec toute l'équipe de ce MOOC nous éprouvons une grande fierté à vous guider dans cette aventure.

Dans la suite de cette activité introductive, nous allons présenter quelques notions liées au fonctionnement d'un ordinateur. Nous commencerons par définir le rôle du système d'exploitation et en particulier celui de type Unix. Nous reviendrons ensuite de façon plus technique sur les fonctions du shell. Nous expliquerons pourquoi la version Bash du shell est celle retenue dans ce MOOC. Puis nous terminerons cette activité en justifiant l'utilité d'un langage pour une interface shell.

2 Le système d'exploitation de type Unix

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

Un ordinateur est en ensemble d'éléments électroniques et mécaniques animé par un calculateur central appelé micro-processeur. Sur cet ensemble se greffent des composants plus ou moins « externes », dits périphériques, qui sont des dispositifs dédiés aux entrées et sorties de données vers et depuis le micro-processeur. Certains de ces périphériques sont en charge de l'interaction avec l'utilisateur comme par

exemple le haut-parleur, le clavier ou encore l'écran. Pour chacun de ces types de périphériques, il y a de nombreux modèles avec pour chacun d'eux des spécificités techniques propres.

L'utilisation de ces équipements depuis un programme informatique demanderait au concepteur de ce programme (le programmeur) d'écrire du code qui va interagir directement avec les équipements. Mais c'est là un travail complexe, car il faudrait prendre en compte toutes les diversités techniques pour un même type d'équipement. Car la portabilité d'un tel code (le fait d'assurer son bon fonctionnement pour tous les cas de figure) obligerait à prendre en compte, dans le développement, tous les périphériques qui pourraient être rencontrés. Ce serait là une tâche beaucoup trop longue et compliquée, et le résultat serait un programme dont le code serait volumineux avec le risque d'un nombre important de bugs. Aussi, la solution proposée à cette problématique est ce qu'on appelle le système d'exploitation.

Logiciel système

Un logiciel système est un groupement de programmes et de bibliothèques logicielles qui forment un support permettant de créer et de faire fonctionner d'autres logiciels qualifiés de « logiciel applicatifs ».

Le système d'exploitation est un logiciel système qui joue un rôle d'intermédiaire entre les programmes et les éléments matériels de l'ordinateur. C'est le premier programme exécuté au démarrage de la machine, et c'est le seul qui reste en permanence en exécution. On le nomme également noyau (*kernel*).

Le rôle du noyau est de masquer l'hétérogénéité des composants en offrant des fonctionnalités de manipulation uniformes, quel que soit le dispositif réellement connecté à l'ordinateur. Il décharge ainsi les programmes de la gestion spécifique de chaque matériel. Le noyau gère notamment les entrées sorties (I/O : *Inputs/Outputs*), que ce soit vers ou depuis les périphériques de stockage, le clavier ou encore l'écran. C'est aussi un gestionnaire de ressources qui s'occupe de distribuer la mémoire, le temps de calcul CPU, les accès aux périphériques, etc. de manière équitable et efficace entres les différents programmes qui s'exécutent en même temps. La vision simplifiée que l'on peut avoir d'un ordinateur peut se représenter par la figure 0.3.1. L'utilisateur accède à des applications qui utilisent des ressources matérielles qui sont gérées et contrôlées par le noyau. Rien ne peut se faire sans le concours du noyau. C'est la raison pour laquelle l'ordinateur se bloque lorsqu'il y a une erreur dans le noyau.

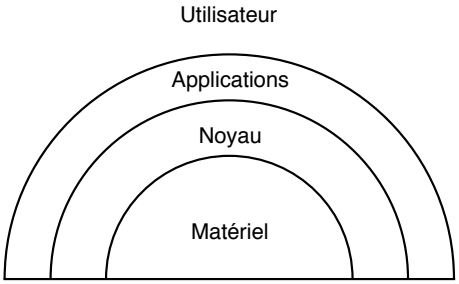


FIGURE 0.3.1 – Organisation d'un ordinateur.

2.2 Un système d'exploitation nommé Unix

Arrière plan

On dit qu'un programme s'exécute en arrière plan lorsqu'il n'a pas besoin d'être associé à une session. Autrement dit, il n'a pas besoin d'un écran et d'un clavier pour fonctionner. Nous reviendrons sur cette notion dans la suite de ce cours.

Le système d'exploitation Unix est un noyau très complet. En plus de prendre en charge la gestion de

la mémoire, celle des entrées/sorties et de la répartition des temps de calcul entre les programmes, il incorpore également des programmes utilitaires pour effectuer des tâches courantes comme la lecture des fichiers ou leur édition. Ces programmes peuvent être sollicités à la demande, ou bien encore s'exécuter en permanence en arrière plan (on parle alors de service) de sorte à toujours être disponibles instantanément. Cet ensemble de constituants est accompagné de fichiers de configuration spécifiques qui définissent les nombreux réglages de ces différents utilitaires et services. C'est cet ensemble composé du noyau et des programmes utilitaires qu'on appelle le système Unix.

Le système d'exploitation Unix peut se représenter sous la forme de couches comme le montre la figure 0.3.2. Au sommet de ces couches se trouve l'utilisateur qui va utiliser l'ordinateur pour atteindre des buts. Pour cela, il va interagir avec des programmes. Ces programmes vont interpréter les actions qu'il réalise sur les périphériques d'entrées comme le clavier et retourner des états sur les périphériques de sorties comme l'écran. Ces programmes peuvent être :

— des commandes : l'ensemble des programmes utilitaires fourni avec le système,
— le shell : un programme spécifique pour une interaction en mode texte pour accéder aux commandes et aux fonctions du noyau,
— des applications spécifiques aux besoins de l'utilisateur (interface graphique, traitement de texte, navigateur web, etc.).

Nous allons maintenant détailler le rôle du shell dans l'interaction avec l'utilisateur.

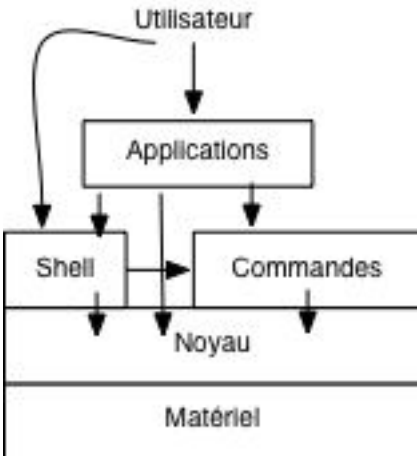


FIGURE 0.3.2 – Représentation simplifiée du système d'exploitation Unix.

3 Le shell

Le shell ou coquille en français est la couche logicielle pour accéder aux fonctions du noyau et pour accéder aux commandes du système d'exploitation. La philosophie des systèmes d'exploitation de type Unix est de privilégier une diversité de petits programmes utilitaires spécialisés et optimisés, plutôt que des programmes volumineux capables de « tout faire ». Mais « tout faire » est devenu possible à l'aide du shell. En effet, le shell donne à l'utilisateur les moyens de manipuler tous les programmes, et même de les assembler pour accomplir des tâches complexes. Il en résulte un procédé bien plus flexible et efficace, et c'est ce qui explique que le shell occupe encore aujourd'hui une place importante dans le paysage informatique moderne.

3.1 Une coquille à plusieurs fonctions

Le shell se présente sous la forme d'une interface en ligne de commande, accessible depuis une console, qu'on appelle aussi un terminal. L'utilisateur exécute des commandes sous la forme d'une ligne de texte

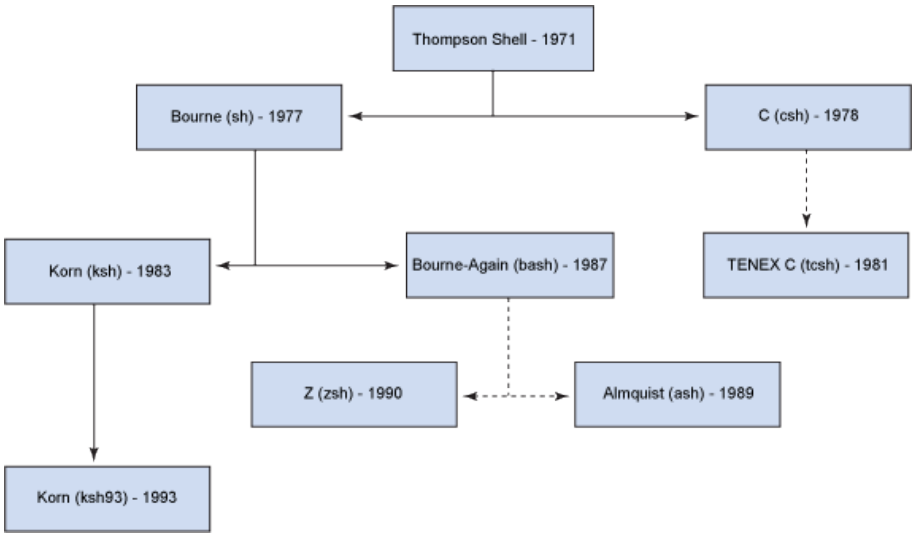


FIGURE 0.3.3 – Généalogie des shells.

dont le shell se charge d'interpréter la syntaxe. Le shell est ainsi une application qui sert d'interface entre l'utilisateur et le système d'exploitation Unix.

En fait, le shell exécute les commandes que l'utilisateur saisit dans la console, c'est ce qu'on appelle le mode interactif. Mais il peut aussi traiter une liste de commandes pré-enregistrées dans un fichier ; on parle alors de mode script, ou plus généralement de script shell. Un script shell peut être utilisé pour créer des applications en combinant les fonctions du noyau, les utilitaires et les applications. Et comme si cela ne suffisait pas, le shell comporte des structures de programmation qui apportent une flexibilité supplémentaires aux scripts. Il devient alors un langage de programmation assez complet. Les scripts shell sont particulièrement adaptés aux tâches administratives du système et à d'autres tâches répétitives ne nécessitant pas d'avoir recours à des développements avec des langages de programmation de haut niveau.

Le shell est donc une interface par laquelle l'utilisateur interagit avec l'ordinateur. D'un point de vue fonctionnel, il joue ainsi le même rôle que l'interface graphique d'un système Windows, Linux, ou Mac OS. Mais, comme nous allons le voir dans ce MOOC, malgré le fait qu'il soit moins raffiné visuellement, sa puissance ne connaît aucune limite !

Pour terminer, notons qu'un terminal et une console peuvent être confondus fonctionnellement, ils proposent tous les deux d'effectuer une session en shell. Et, de ce point de vue, on ne peut rien faire de plus ou de moins qu'on soit dans une console ou un terminal. En pratique, la console est l'équipement physique d'un terminal sans mode graphique. C'est une sorte de Minitel (pour ceux qui ont connu). À ce titre, il n'a pas de barre de défilement sur le côté au contraire d'un terminal. Un terminal prend de nos jours la forme d'une application en mode fenêtre dans un environnement graphique pour émuler une console.

3.2 Le shell Bash

Même s'il joue le rôle aussi spécifique que celui que nous venons de voir, il n'en reste pas moins que du point de vue du système Unix, le shell n'est autre qu'un programme utilitaire parmi d'autres. Par conséquent n'importe qui peut créer son propre shell du moment que celui-ci est conforme au rôle attendu d'un shell. Cela a mené à l'apparition d'une douzaine de shells concurrents dans les années 90.

Les shells se différencient par leur facilité d'utilisation, leur programmabilité ainsi que par leur licence d'utilisation. Ceux qui sont conformes à la norme POSIX présentent le plus d'avantages, notamment

en matière de portabilité. Car ils partagent un même ensemble de syntaxe, ce qui assure notamment que des commandes ou des scripts développés dans l'un des shells respectant cette norme pourra fonctionner à l'identique dans tous les autres shells conformes à POSIX.

Norme POSIX

POSIX est une famille de normes techniques définie depuis 1988 par l'Institute of Electrical and Electronics Engineers (IEEE), et formellement désignée par IEEE 1003. Ces normes ont émergé d'un projet de standardisation des interfaces de programmation des logiciels destinés à fonctionner sur les variantes du système d'exploitation Unix. (source : wikipedia)

Le shell Bash a été conçu comme une mise en œuvre *open source* d'un shell conforme à POSIX. Il inclut également des facilités appréciées par les programmateurs de scripts et les utilisateurs en ligne de commande. Il est donc devenu le shell par défaut des systèmes de type Unix et il est même disponible sous Windows 10. C'est pour cette raison que ce cours se concentre sur le shell Bash. Ce que vous apprendrez dans ce cours pourra donc être appliqué à tous les ordinateurs, et même aux smartphones ou aux tablettes fonctionnant sous Android.

4 Conclusion

Le fait d'utiliser un shell revient à effectuer une interaction avec l'ordinateur par un terminal. C'est-à-dire entrer des commandes à l'aide du clavier. De nos jours, où tous les ordinateurs ont la puissance nécessaire pour offrir une interface graphique, cela peut sembler archaïque d'utiliser un terminal. En fait il n'en est rien, car l'interaction par un langage textuel offre une richesse et une flexibilité sans limite pour décrire des actions.

Open source

Un logiciel open source est un programme informatique dont le code source est distribué sous une licence permettant à quiconque de lire, modifier ou redistribuer ce logiciel. La diffusion et la modification du logiciel sont libres.

Pour s'en convaincre on peut faire l'analogie suivante avec le langage humain : l'interface graphique est l'équivalent d'un petit livre « guide du voyageur », qui contient des phrases toutes faites pour une langue étrangère. Mais connaître la langue étrangère reste le moyen le plus efficace pour faire des phrases adaptées à tous ses besoins de communication, présents et futurs, donc même les besoins auxquels on ne s'attendait pas initialement et qui ne sont pas non plus prévus dans notre petit guide.. . .

Eh bien, maîtriser le shell, c'est en quelque sorte maîtriser un langage pour communiquer avec un ordinateur. Si vous souhaitez être efficace avec l'outil informatique, vous savez maintenant ce que vous avez à faire.

Activité 0.4

Guide d'utilisation de la console Weblinux

Dans ce cours, nous vous proposons d'utiliser une console pour compléter votre apprentissage du shell par la pratique. Cette console s'exécute dans votre navigateur web. Pour rendre cela possible, le navigateur web charge un système Linux qui lance ensuite directement une console avec le shell Bash. Ce système d'exploitation dans le navigateur n'est pas aussi complet qu'un système Linux installé nativement sur un ordinateur. Cependant ce système que l'on appelle Weblinux reprend les principales caractéristiques d'un système d'exploitation de type Unix. L'objectif est de vous donner accès à un shell Bash et ce, quelque soit le système d'exploitation de votre ordinateur. Avec le shell Bash disponible depuis la Weblinux, vous avez les moyens de reproduire facilement les manipulations mentionnées dans les vidéos ou les exercices. Par la suite, vous pourrez vous lancer, sans hésiter, sur un vrai système.

Ce document se propose de vous expliquer le fonctionnement et l'usage de la Weblinux. La section 1 présente cet outil. La section suivante indique comment trouver le lien à la Weblinux du cours. Le guide utilisateur de la console Weblinux est décrit dans la section 3. Malgré sa ressemblance à un système Linux, il existe quelques limitations dans la Weblinux rappelées dans la section 4. Enfin, il se peut que vous rencontriez des problèmes. La section 5 peut vous aider dans ce cas-là.

1 Présentation de la Weblinux

Ce cours vous offre un environnement d'apprentissage commun sans nécessité d'installation sur votre ordinateur. La Weblinux se télécharge et fonctionne sur n'importe quel navigateur web. Cette solution reste très légère et économique. En effet, il n'est pas nécessaire de procéder à une installation de Linux sur votre ordinateur. Weblinux vous donne l'équivalent d'un ordinateur virtuel Linux simplifié.

Une fois téléchargée, la Weblinux démarre et ouvre une console dans une fenêtre de votre navigateur. Notez qu'une fois chargée dans votre navigateur, cette Weblinux s'exécute localement dans votre navigateur et reste interne à votre ordinateur. La console de la Weblinux se présente selon la figure 0.4.1.

Afin de satisfaire la configuration d'un maximum d'utilisateurs, la résolution minimale de la fenêtre du navigateur de la Weblinux est de 800x600 pixels. La taille de l'écran de la console est quant à elle fixée à 26 lignes et 90 caractères.

Le démarrage de la Weblinux consiste à exécuter un noyau Linux dans **votre** navigateur. C'est pour cette raison qu'une liste de messages s'affiche sur la console et qu'un certain temps est pris avant que vous ne puissiez utiliser la console. Enfin, un système de fichiers est constitué dans la mémoire du navigateur. Il faut savoir que les fichiers que vous pourriez créer ne sont pas conservés d'un démarrage à un autre de la Weblinux. Il vous faut les sauvegarder sur votre ordinateur. Cette manipulation est expliquée dans la section 3.

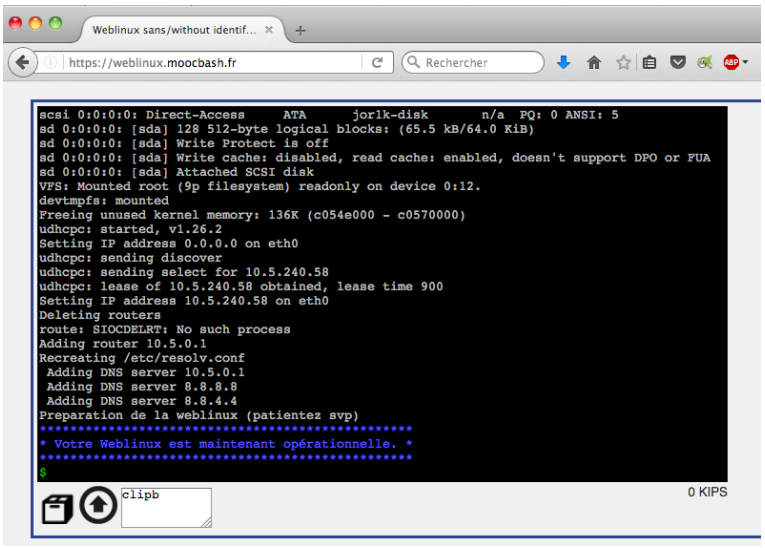


FIGURE 0.4.1 – Console de la Weblinux dans une fenêtre d'un navigateur web.

Une fois chargée, la Weblinux reste en grande partie dans le cache de votre navigateur. Les utilisations successives de Weblinux font que le système Weblinux se relance rapidement (d'où la qualification « économique » en ressources de communication).

2 L'accès à la Weblinux

Concrètement, vous trouverez le lien pour démarrer la Weblinux à partir du menu général (bandeau supérieur) du cours sous l'onglet Weblinux. Vous pouvez aussi indiquer ce lien explicitement dans votre navigateur. Vous serez identifié en tant qu'utilisateur *alice*. Nous reviendrons sur cette notion d'identifiant utilisateur dans l'activité 1.4. La Weblinux ne requiert pas d'authentification et vous permet d'accéder directement au shell Bash.

Dès l'ouverture de la nouvelle fenêtre, et dès le premier téléchargement, le navigateur récupère le noyau de la Weblinux ainsi que quelques fichiers d'initialisation. Le temps de démarrage de la Weblinux est lié au temps de transfert et au temps d'exécution. Ces temps dépendent respectivement du débit de votre connexion Internet et de la puissance de votre ordinateur. Une grande partie du code de la Weblinux est conservée dans le cache de votre navigateur. À partir du second téléchargement, le temps de transfert diminue, ce qui a pour conséquence de rendre le redémarrage beaucoup plus rapide. Lorsque un caractère `'$'` vert apparaît, cela vous informe que la Weblinux est désormais opérationnelle et que c'est à vous de jouer.

3 Tutoriel pour l'usage de la Weblinux

3.1 Les boutons de l'interface de la Weblinux

Dans la partie inférieure de la fenêtre de la Weblinux, vous trouverez 3 boutons comme le montre la figure 0.4.2. Ces boutons sont, de gauche à droite :

— une petite boite ; ce bouton sert à télécharger le contenu du répertoire `alice/shared`. Par défaut, ce répertoire est vide, vous mettez dedans les fichiers que vous voulez sauvegarder et donc télécharger sur votre ordinateur. Attention cependant à ne pas mettre des grosses quantités de données, cela peut charger le traitement de la weblinux et vous aurez l'impression qu'elle est bloquée. Donc ce bouton de téléchargement est destiné pour mettre quelques fichiers et

FIGURE 0.4.2 – Boutons de la fenêtre Weblinux.

typiquement à récupérer vos fichiers écrits dans la weblinux.
Ce bouton lorsqu'il est activé crée une archive `user.tar` avec le contenu du répertoire `alice/shared` puis transfère cette archive sur votre ordinateur. Il vous reste alors à traiter l'archive `user.tar` sur votre ordinateur. En ligne de commande, vous apprendrez qu'il faut faire la commande `tar xvf user.tar` dans le répertoire de cette archive pour extraire les fichiers de l'archive.

— la flèche vers le haut ; ce bouton représente la fonction d'envoyer (*upload*). Il sert à transférer un fichier depuis votre ordinateur vers Weblinux. Vous vous en servirez pour restaurer votre travail (que vous avez préalablement sauvegardé).

— un rectangle blanc ; ce rectangle représente un presse-papier (*clipboard*). Le texte issu du presse-papier de votre ordinateur sera collé comme une saisie dans la console de la Weblinux. Pour faire cette opération de copier-coller, vous devez tout d'abord copier le texte à partir d'une fenêtre de votre ordinateur. Puis, vous devez cliquer sur le rectangle blanc de la fenêtre de la Weblinux et à l'aide d'un clic secondaire, lorsque le menu apparaît, choisir l'action *coller* comme indiqué par la figure 0.4.3.

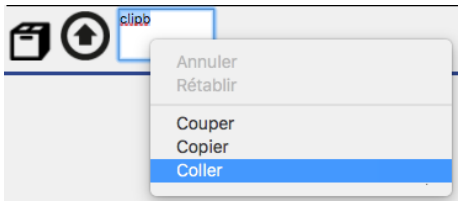


FIGURE 0.4.3 – Coller dans la fenêtre Weblinux.

À droite de ces boutons, apparaît l'usage des touches de fonctions de votre clavier, tel des voyants d'activité. L'usage des touches Shift, Control et Alt est affiché.

3.2 Entrer des caractères spécifiques à un système de type Unix

Au fil des activités de ce cours, vous allez découvrir des caractères qui ont des significations spéciales et particulières à Unix. Ces caractères ne sont pas toujours indiqués explicitement par des touches de votre clavier. Nous vous présentons ici comment les obtenir.

Clavier Apple en français

Certains caractères usuels des systèmes d'exploitation de type Unix tels que le *pipe* (|) ou le *tilde* ('~') s'obtiennent avec des combinaisons de touches. Le tableau ci-dessous précise la combinaison de touches pour ces caractères.

Caractères	Combinaisons de touches
~	Alt + n
|	Alt + Shift + L
[	Alt + Shift + (
]	Alt + Shift +)
{	Alt + (
}	Alt +)
\	Alt + Shift + /
^	^ + Space

La correspondance des touches réalisée par Chrome, Safari ou Opéra impose que le caractère '|' se fasse avec la combinaison de touches Ctrl + Shift + L. Alors qu'avec Firefox et comme c'est le cas dans un terminal, le caractère '|' s'obtient avec la combinaison Alt + Shift + L.

L'accès à ce caractère spécial '|' est indiqué sur l'interface de votre navigateur lorsque cela est utile (en bas à droite).

Clavier PC en français

Les caractères *pipe* ('|'), *tilde* ('~') ou *backquote* ('`') sont obtenus avec l'usage de la touche AltGr située à droite de la barre d'espacement combinée avec une autre touche comme le montre le tableau ci-dessous :

Caractères	Combinaisons de touches
~	AltGr + 2
|	AltGr + 6
[	AltGr + 5
{	AltGr + 4
]	AltGr +)
}	AltGr +
\	AltGr + 8
^	AltGr + 9
`	AltGr + 7

4 Les limites de la Weblinux

La Weblinux a la couleur et le goût du Linux, mais ce n'est pas du Linux. L'objectif de la Weblinux est d'offrir une interface en ligne de commande utilisable dans un navigateur. Elle permet de découvrir ce mode d'interaction mais elle n'a pas la puissance d'un vrai système. Elle est très suffisante pour vos premières commandes en Bash.
On pourrait faire l'analogie avec un décor de cinéma ; tout est dans l'apparence car il existe de nombreuses différences entre la Weblinux et une distribution Linux classique. Nous pouvons citer les différences suivantes :

— Le nombre de commandes. Pour alléger le chargement de la Weblinux, toutes les commandes d'un système de type Unix ne sont pas disponibles. En effet, nous avons supprimé de nombreuses commandes non-pertinentes dans le cadre d'un apprentissage du shell.
— Le manuel en ligne appelé couramment le *man*. Nous avons décidé de mettre à disposition le

manuel des commandes Linux, en français [1]. Cette aide en ligne explique, en français le plus souvent, le comportement et la syntaxe des commandes disponibles dans un système Linux classique. Nous reviendrons en détail sur l'aide en ligne lors de la première séquence.
Pour retrouver l'usage réel des commandes disponibles, il faut faire suivre le nom de la commande par l'option `--help`. Par exemple, si vous tapez `mkdir --help`, vous obtiendrez l'aide de la commande `mkdir`. Notez que cette aide succincte est en anglais.

— La syntaxe de certaines commandes. Celle-ci peut différer légèrement par rapport à la description faite dans le manuel ou par rapport à la version d'un Linux classique.

— Le système de fichiers. Celui-ci est émulé en Javascript et créé au moment du téléchargement de la Weblinux dans le navigateur. De ce fait, tous les fichiers portent la même date et il y a parfois un bug lors de la suppression de fichiers. De facto, la manipulation des dates de fichiers ne fonctionnent pas comme sur un système Linux classique.

5 En cas de problème

Si la Weblinux vous semble bloquée, nous vous suggérons les solutions suivantes :

— Regardez les messages de votre antivirus. Il se peut que l'antivirus empêche l'accès de la weblinux aux fichiers du serveur. Fort heureusement, cela est très rare !

— Tout d'abord vérifier que la fenêtre de la Weblinux est bien la fenêtre courante ou active. Plus précisément que la partie console est la zone active. Pour cela cliquez sur la console ou laissez le pointeur de la souris au dessus de la console en fonction de votre système de gestion de l'interface graphique que vous utilisez.

— Soyez aussi patient : votre navigateur est peut-être très occupé et l'exécution du système Weblinux très ralentie.

— Si malgré toute votre patience rien ne bouge, qu'il ne vous n'est plus possible de saisir des caractères, que le clavier semble inopérant, alors vous pouvez essayer d'interrompre l'exécution en cours par la combinaison de touches CTRL + C (appui simultané sur ces 2 touches).

— La situation n'est toujours pas redevenue à la normale ? Il faut alors avoir recours au grand moyen : il faut redémarrer la Weblinux. Pour cela, il faut recharger la Weblinux soit en actualisant sa fenêtre soit en la fermant, puis en l'ouvrant à nouveau. Il va sans dire que cette solution entraîne la perte de tout le travail effectué lors de cette session. Pour se prémunir de ce désagrément, il peut être pertinent d'effectuer l'édition des fichiers sur son ordinateur puis de les envoyer ensuite sur la Weblinux pour la mise au point et leur exécution.

— Il faut savoir que la Weblinux occupe 95 Mo dans sa totalité. En fait, le noyau a une taille 2.2 Mo et les principaux fichiers initiaux 3 Mo. Ensuite chaque nouvelle commande est chargée dynamiquement. Ainsi, lors du premier chargement de la Weblinux, celle-ci est stockée dans le cache du navigateur. Le redémarrage de la Weblinux par le rechargement revient à recharger la page web de présentation et quelques scripts en Javascript pour moins de 1 Mo et ensuite à utiliser le cache du navigateur. Pour se retrouver dans la situation initiale ou pour charger la dernière version de Weblinux, il faut préalablement vider le cache du navigateur. La procédure de vidage du cache du navigateur est propre à chaque navigateur. Vous trouverez sur Internet ou dans l'aide de votre navigateur les manipulations à effectuer.

— Bien que les messages d'erreur puissent être difficiles à comprendre, affichez la console de votre navigateur (souvent dans les outils de développement – selon les navigateurs –), vous y trouverez des messages précis qui indiqueront l'origine de votre problème.

Quoi qu'il en soit, nous avons ouvert un forum dédié à tous les problèmes que vous pourrez rencontrer dans l'utilisation de la Weblinux. N'hésitez pas à nous signaler les éventuels défauts.

1. extrait de la distribution Ubuntu 16.04

6 Conclusion

L'outil proposé vous permet d'exécuter un Linux dans votre navigateur. Nous espérons qu'il répondra à vos attentes et qu'il vous aidera dans l'apprentissage du shell Bash. Vous pouvez maintenant utiliser le Bash via la Weblinux et saisir des commandes. Mais n'anticipons pas, tout cela vous sera expliqué à partir de la première séquence.

Si vous avez encore des questions, n'hésitez pas à consulter le forum dédié à l'utilisation de la Weblinux mis à disposition spécifiquement pour cela.

Annexe A : Spécifications techniques

Le noyau Linux est exécuté dans le navigateur, des scripts Javascript émulent les périphériques (carte réseau, périphérique d'entrée ou de sortie, carte graphique, clavier, etc...). La Weblinux est construite sur la base d'un émulateur Javascript pour processeur OpenRISC 1000 (abrégé en jOR1k). Cet outil est développé principalement par M. Sebastian Macke dans le cadre général d'un projet d'émulateur de microprocesseur et disponible gratuitement à cette adresse :
`https://s-macke.github.io/jor1k/demos/main.html`
Nous le remercions d'ailleurs pour avoir rendu disponible le code de cet outil en *open source*.

Séquence 1

Découvrez votre système d'exploitation

Introduction

Cette séquence constitue votre première rencontre avec le Bash. Comme toutes les séquences de ce cours, celle-ci étudie 5 thèmes, aussi appelés activités :

1. Dans la première activité nous allons découvrir ce que sont un terminal, le Bash et la notion de commande pour exécuter des programmes.
2. Dans la seconde activité nous allons vous rendre autonome, vous serez capable de trouver par vous-même de l'aide sur les commandes du Bash.
3. La troisième activité vous permettra de comprendre le système de fichiers de votre ordinateur, de l'explorer, de le gérer et enfin d'en tirer profit.
4. Dans la quatrième activité vous découvrirez la gestion des droits du système de fichier. Vous serez à même de protéger vos fichiers afin que les autres utilisateurs n'y aient pas accès.
5. Enfin, la cinquième activité sera consacrée aux fichiers texte et nous verrons comment les consulter et les éditer de manière efficace.

Ces notions constituent le prérequis à l'utilisation du Bash. Si vous parvenez à la fin de cette séquence, vous serez à même d'utiliser le Bash pour lancer des commandes et interagir avec le contenu de votre système de fichiers. C'est largement suffisant pour une utilisation basique de votre terminal.

Il n'y a plus qu'à commencer !

Activité 1.1

La ligne de commande

1 Introduction

Après avoir défini le rôle du shell lors de la première semaine, nous allons maintenant découvrir la ligne de commande. Comme vous le savez maintenant, le shell désigne un interpréteur de lignes de commande. Cette dernière est l'unité d'interaction avec l'utilisateur. Cette interaction est faite par du texte et elle est associée à la notion de terminal. C'est par cette modalité que l'utilisateur peut exprimer l'action qu'il souhaite réaliser.

Lorsque l'invite de commande s'affiche, cela signifie que l'utilisateur peut saisir une nouvelle ligne de commande. La ligne de commande regroupe une ou plusieurs commandes et se termine par un retour à la ligne. Une commande est composée d'un nom qui décrit de façon mnémonique une action ou un programme, parfois suivie d'arguments qui précisent les paramètres de l'action à effectuer.

```
$ cal -m apr
```

L'exemple ci-dessus est une ligne de commande qui ne contient qu'une seule commande. Le nom de la commande est `cal` (diminutif de calendrier) suivi de deux arguments `-m` (diminutif de mois) et `apr` (diminutif de *April*). Avec seulement quelques caractères à saisir, cette commande effectue l'action d'afficher le calendrier du mois d'avril. Vous en conviendrez, c'est plutôt efficace.

Notion de répertoire

Le répertoire (aussi appelé dossier) fait référence au système de fichiers de votre ordinateur. Votre shell Bash est en permanence associé à un répertoire dans lequel s'exécutent les commandes. Ce répertoire est appelé répertoire courant (voir l'activité 1.3).

Après avoir introduit l'invite de commande dans la section 2, nous détaillerons les notions de commande et ligne de commande dans la section 3. Nous y verrons également l'algorithme du shell, c'est-à-dire la suite d'actions génériques qui suit l'appel à une commande. Ces notions seront illustrées à travers des commandes classiques telles que `echo`, `date` et `cal`.

2 Invite de commande

Lorsque vous démarrez un terminal sur lequel le shell Bash est présent, vous êtes accueilli avec l'invite de commande appelée *prompt* en anglais. Elle vous indique que le terminal est prêt à accepter votre ligne de commande. Cette invite de commande apparaît également chaque fois que la précédente commande a fini son exécution. Son format et sa signification peuvent varier en fonction des configurations mais la plupart du temps elle est similaire au format suivant :

"login"`@`*"nom d'hôte"*`:` *"répertoire courant"*`[$|#]`

L'identifiant de l'utilisateur correspond à l'identifiant du compte shell associé (on parle de *login*). Les commandes vont être exécutées pour le compte de l'utilisateur et avec les droits de cet utilisateur. Le nom d'hôte (ou nom de machine) indique la machine sur laquelle les commandes sont exécutées.

```
alice@pc-alice.boulot.fr:~$
```

Dans l'exemple ci-dessus, l'utilisateur sait instantanément qu'il travaille avec l'identifiant *alice* et que le shell de ce terminal s'exécute sur la machine *pc-alice.boulot.fr*. Cela peut être particulièrement utile lorsque l'utilisateur est connecté à une machine distante (par `ssh`). Cela lui évite d'insérer des commandes dans le mauvais terminal.

Session ssh

La commande ssh (*Secure SHell*) donne l'accès au shell d'une autre machine. Ainsi, Alice peut se connecter à la machine de son travail depuis son domicile avec la commande `ssh` suivante :
`ssh alice@pc-alice.boulot.fr`

Le répertoire courant indique dans quel répertoire sera exécutée la commande. Par exemple si vous demandez à lister le contenu d'un répertoire, il est préférable de savoir si on se trouve dans le bon répertoire. Cependant, le nom de répertoire est souvent abrégé pour éviter qu'il occupe trop d'espace sur la ligne de commande. Ainsi, le répertoire personnel de l'utilisateur (en anglais *home directory*) est habituellement abrégé par le caractère tilde noté `'~'`.

L'invite de commande peut se terminer par l'indication du type d'utilisateur. S'il s'agit d'un utilisateur normal, l'invite se terminera par le caractère dollar `$`. S'il s'agit du super-utilisateur (son identifiant est *root*), la commande se terminera par le caractère `#`. Avec l'invite de commande, en un coup d'œil le contexte d'exécution de la commande est tout de suite connu. Cette désambiguïsation réduit les chances de se tromper. L'activité 1.4 reviendra sur les droits des utilisateurs et du super-utilisateur.

Par exemple, l'invite de commande ci-dessous nous indique que la commande sera exécutée en tant qu'utilisateur *tonton*, sur la machine ayant pour nom d'hôte *portable1*, dans le répertoire personnel de tonton. La présence du suffixe $ nous rappelle que la commande sera exécutée avec les droits de l'utilisateur *tonton*.

```
tonton@portable1:~$
```

L'invite de commande ci-dessous nous indique que la commande sera exécutée en tant que super-utilisateur, sur la machine ayant pour nom d'hôte *portable1*, dans le répertoire `/tmp/abc/def/`. La présence du suffixe # nous rappelle que la commande sera exécutée avec les droits du super-utilisateur.

```
root@portable1:/tmp/abc/def/#
```

3 Commande

Comme nous l'avons vu, le shell est une application qui sert d'interface entre le noyau et l'utilisateur. Il sert à exécuter des commandes qui proviennent d'un terminal (mode interactif) ou d'un fichier (mode script). Ces commandes peuvent être internes au shell ou externes au shell [1]. Les commandes externes font appel à des programmes séparés du shell tandis que les commandes internes sont exécutées par le shell lui-même.

1. Ces commandes externes peuvent être des programmes fournis par le système d'exploitation ou par l'utilisateur lui-même.

Il est possible de savoir de quel type est une commande. La commande interne `type` suivie du nom d'une commande sert à indiquer le type de la commande.

```
$ type echo man date
echo is a shell builtin
man is /usr/bin/man
date is hashed (/bin/date)
```

Ici, `type` nous apprend que la commande `echo` est de type *built-in*, c'est-à-dire une commande interne. La commande `man` est une commande externe dont le programme est dans le répertoire `/usr/bin`. La commande `date` est également une commande externe mais `type` nous indique que cette dernière est « hashée », c'est-à-dire que la localisation du programme a été mise en mémoire afin d'accélérer le lancement de la commande.

✍ **Challenge C11Q1**

Type des commandes

3.1 Algorithme du shell

Une commande est généralement un appel à un programme. Après avoir saisi la ligne de commande, il suffit d'appuyer sur la touche de retour à la ligne pour lancer son exécution. Avant l'exécution, le shell effectue l'analyse de la ligne de commande pour y retrouver les éléments constitutifs. Nous reviendrons en détail sur cette étape dans la séquence 2. Pendant l'exécution, l'utilisateur peut être sollicité pour entrer des données supplémentaires. Une fois l'exécution de la commande terminée, le résultat s'affiche à l'écran et l'invite de commande s'affiche à nouveau, ce qui indique que le shell est prêt à recevoir une nouvelle ligne de commande.

✍ **Challenge C11Q2**

Algorithme du shell

3.2 Structure d'une commande

Une commande (interne ou externe) est constituée par des mots séparés par des espaces. Le format général d'une commande est le nom de la commande suivi d'arguments qui seront interprétés avec cette dernière. Les arguments sont passés avec l'appel du programme associé à la commande.

```
$ commande arg1 arg2 ... argn
```

Le nombre d'arguments dépend de la commande et de l'action à effectuer par la commande. Par exemple la commande `date` peut être appelée sans aucun argument. Elle affichera alors la date :

```
$ date
jeudi 23 mars 2017, 06:57:11 (UTC+0000)
```

On peut spécifier à la commande `date` que l'on veut une date affichée comme le nombre de secondes écoulées depuis le 1e premier janvier 1970 :

```
$ date +%s
1490252400
```

Parmi les commandes les plus simples, on retrouve `echo` dont le but est uniquement d'afficher les mots qui lui sont passés en arguments. L'espace joue un rôle de séparateur de mots. Pour passer un argument qui contient des caractères espace, il faut l'encadrer par le caractère guillemet (*double quote)*. L'exemple ci-dessous montre l'affichage d'un seul argument sous la forme de la chaîne de caractères. A noter que les guillemets ne sont pas affichés.

```
$ echo "Bonjour le monde"
Bonjour le monde
```

✍ **Challenge C11Q3**

Appréhender la commande `echo`

La commande `cat` est une autre commande simple. Elle est utilisée pour afficher le contenu d'un petit fichier. Nous reviendrons dans l'activité 1.5 sur la présentation de cette commande.

La syntaxe attendue pour chaque argument est spécifique à chaque commande. Un argument peut être une option, il sera alors de la forme `-x` avec x la lettre identifiant l'option. Une lettre étant peu explicite, il est souvent possible d'identifier une option via un ou plusieurs mots. Elle sera alors préfixée de deux `--`. Par exemple `date -u` et `date --utc` font référence à la même option, `--utc` ayant l'avantage d'être plus explicite que `-u`.

```
$ date -u
Mon Feb  5 04:18:00 UTC 2018
$ date --utc
Mon Feb  5 04:18:42 UTC 2018
```

Pour qu'elle ait un sens, une option doit parfois être suivie d'une valeur, appelée argument d'option. Cette valeur est spécifiée dans l'argument qui suit l'option. Par exemple on peut demander à la commande `cal` (*CALendar*) d'afficher le calendrier du mois d'avril (*april* en anglais) via la commande suivante (option `-m` avec l'argument d'option `apr`) :

```
$ cal -m apr
      April 2018
 Mo Tu We Th Fr Sa Su
                    1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
 30
```

La valeur associée à l'option peut être spécifiée dans le même argument mais séparée de l'identifiant d'option via un caractère délimiteur. Par exemple, pour demander à la commande `date` de modifier le format d'une date passée en argument, le caractère délimiteur entre l'option et sa valeur sera le `'='` comme le montre la commande suivante :

```
$ date --date="2004-02-29"
Sun Feb 29 00:00:00 GMT 2004
```

✍ **Challenge C11Q4**

Ligne de commande

Normalisation des commandes

Les options proposées par une commande dépendent de sa version. Chaque système d'exploitation met en oeuvre sa propre version. Ainsi l'interface, c'est à dire les options, d'une commande peuvent être différentes d'un système Unix à un autre. Ceci pose des problèmes à l'utilisation d'Unix et plus généralement de portage des scripts shell entre les différents systèmes Unix. La norme POSIX (*Portable Operating System Interface*) répond à ce besoin de compatibilité entre les systèmes d'exploitations qui se trouvent sur les différents équipements informatiques. Elle définit diverses interfaces d'outils, de commandes et d'interfaces pour la programmation en langage C. Cette norme a été reprise par l'Open Group qui a travaillé à la certification d'un système d'exploitation pouvant s'annoncer comme étant Unix. Pour cela, elle a publié un ensemble de spécifications connu sous le nom de SUS (*Single UNIX Specification*). A noter, L'Open Group possède la marque de commerce Unix.

4 Conclusion

Dans cette activité, vous avez découvert la ligne de commande. Vous connaissez maintenant le mode interactif du Bash. Lorsqu'une invite de commande s'affiche, cela signifie que vous pouvez entrer une ligne de commande. Une fois validée, elle est interprétée par le shell et vous devez attendre que la commande termine son exécution pour que l'invite de commande s'affiche à nouveau.

Nous avons également vu la structure d'une ligne de commande. Elle est composée d'un nom suivi d'options et d'arguments. Notez cependant que la commande a une syntaxe qui lui est propre. Il est donc indispensable d'obtenir de l'aide pour savoir l'utiliser. C'est l'objet de la prochaine activité.

Activité 1.2

Trouver de l'aide

1 Introduction

Les arguments attendus par les commandes du shell et les fonctionnalités fournies sont trop diversifiés pour être retenus par l'utilisateur ou réunis dans un manuel unique. Par conséquent, la documentation est fournie par les commandes elles-mêmes et l'utilisateur peut y accéder soit au travers de la commande, soit au travers d'utilitaires dédiés.

Lorsque vous n'êtes pas sûr du nom de la commande à utiliser pour accomplir une action, le système propose également des outils pour vous aider à trouver la commande adéquate.

2 Accéder à l'aide depuis la commande

La plupart des commandes disposent d'une aide interne. Lorsque l'utilisateur saisit de mauvais arguments, la commande invite à consulter cette dernière :

```
$ date --nimportequoi
date: unrecognized option '--nimportequoi'
Try 'date --help' for more information.
```

Dans le cas de la Weblinux, l'aide s'affiche directement à la suite du message d'erreur. Dans le cas général, suite à cette erreur d'option, la commande suggère d'utiliser l'option `--help` pour obtenir de l'aide.

```
$ date --help
Usage: date [OPTION]... [+FORMAT]
  or:  date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
Display the current time in the given FORMAT, or set the system date.
(...)
```

Notons que la documentation est alors directement affichée dans le terminal. Il n'est pas aisé de naviguer entre les différentes parties ou même de rechercher du texte à l'intérieur. De plus, l'aide affichée est souvent en anglais et ce même si le système d'exploitation a été installé avec un autre langage.

✍ **Challenge C12Q1**

Aide interne

Remarque :
La Weblinux est une émulation simplifiée d'un système Unix. Les pages du manuel peuvent indiquer des commandes avec une interface plus riche que celles disponibles dans la Weblinux. Les exemples présentés ne sont pas tous extraits de la Weblinux.

3 Accéder à l'aide avec la commande `man`

Les systèmes de type Unix disposent d'un outil de visualisation des manuels appelé `man`, diminutif du terme anglais *MANual*. Lorsque l'on cherche à accéder à la documentation d'une commande, il suffit d'appeler `man` avec pour argument le nom de la commande dont on cherche le manuel. Par exemple, on peut accéder au manuel de la commande `date` par la commande suivante :

```
$ man date
DATE(1)                          User Commands                          DATE(1)

NAME
       date - print or set the system date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
(...)
```

La commande `man` interprète des fichiers de documentation [1] puis les affiche via le lecteur de fichier `less` que nous aborderons dans l'activité 1.5. Sachez simplement que ce lecteur fait défiler le texte par page d'écran et permet de faire des recherches de motifs. Retenez que pour sortir du manuel en ligne, il faut taper la lettre `'q'`.

✍ **Challenge C12Q2**

Accès au manuel

3.1 Langue du manuel

Si votre système d'exploitation est configuré en français, le manuel sera présenté en français s'il existe. Si vous êtes sur un système configuré en anglais et que vous voulez néanmoins accéder à l'aide en français, vous pouvez utiliser l'option `-L fr`. Si le manuel n'existe pas dans la langue demandée, c'est en anglais qu'il s'affichera. L'outil `man` contient lui aussi une page de manuel. La commande suivante permet d'y accéder en français :

```
$ man -L fr man

MAN(1)            Utilitaires de l'afficheur des pages de manuel            MAN(1)

NOM
       man - Interface de consultation des manuels de référence en ligne
```

1. Généralement dans le répertoire `/usr/share/man`.

A noter que cette option n'est pas proposée dans la version standardisée de la commande. Elle ne se trouve pas forcément sur votre Unix comme c'est le cas pour la Weblinux.

✍ **Challenge C12Q3**

Le manuel du manuel

3.2 Sections du manuel

Le manuel du `man` vous apprendra notamment qu'il n'est pas limité aux commandes, il comprend de la documentation sur les fonctions, des formats de fichiers ou encore des périphériques. Par conséquent, les pages du manuel sont regroupées en plusieurs sections :

1. commandes pour l'utilisateur,
2. appels système (fonctions fournies par le noyau),
3. appels de bibliothèque (fonctions fournies pour le développement de programmes),
4. fichiers spéciaux (situés généralement dans `/dev`),
5. formats des fichiers et conventions (par exemple `/etc/passwd`),
6. jeux,
7. divers,
8. commandes de gestion du système (généralement réservées au super utilisateur),
9. documentation du noyau.

Cette distinction a été faite pour éviter les ambiguïtés. En effet un même nom peut faire référence à une fonction d'une bibliothèque de programmation ou à une commande. C'est par exemple le cas de `printf`. Pour accéder à la documentation d'une section en particulier, il faut spécifier le numéro de section en argument. Par exemple `man 1 printf` indique d'accéder à la documentation de la commande `printf`, tandis que `man 3 printf` va accéder à la documentation de la fonction `printf` de la bibliothèque `stdio`.

Dans le cadre de ce MOOC et dans la plupart des cas, vous n'aurez besoin que de la section des commandes de l'utilisateur, c'est-à-dire celle qui s'affiche par défaut.

3.3 Structure d'une page de manuel

Une page de manuel contient plusieurs parties : NOM, SYNOPSIS, DESCRIPTION, OPTIONS, FICHIERS, VERSIONS, NOTES, EXEMPLES, AUTEURS et VOIR AUSSI.

La partie synopsis propose un résumé du format de la commande. Il utilise les conventions suivantes :

```
texte gras          à taper exactement comme indiqué ;
texte italique      à remplacer par l'argument approprié ;
[-abc]              tous les arguments entre [ ] sont facultatifs ;
-a|-b               les options séparées par | ne peuvent pas être
                    utilisées simultanément ;
argument ...      les trois petits points signifient que
                    l'argument peut être répété ;
[expression] ...    les trois petits points signifient que toute
       l'expression située à l'intérieur de [ ] peut être
       répétée (en plus d'être facultative).
```

Ainsi, le synopsis de la commande `date` reproduit ci-dessous nous indique que la commande peut être lancée sans argument, qu'elle peut contenir des options et/ou un format d'affichage.

```
SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMJJhhmm[[CC]AA][.ss]]
```

La signification et l'utilisation des options et arguments sont ensuite détaillées dans la partie DESCRIPTION. Si le synopsis et la description ne sont pas suffisamment explicites, la partie EXEMPLES permettra d'éclaircir la syntaxe à travers des exemples de commandes bien formées.

```
EXEMPLES
       Convertir un nombre de secondes depuis le 01/01/1970 UTC en date
              $ date --date='@2147483647'
       Afficher l'heure de la côte ouest des États-Unis (utilisez tzselect(1)
       pour trouver TZ)
              $ TZ='America/Los_Angeles' date
       Afficher l'heure locale équivalente à 9 heures vendredi prochain sur la
       côte ouest des États-Unis
              $ date --date='TZ="America/Los_Angeles" 09:00 next Fri'
```

Notons que toutes les notions abordées dans ce document peuvent être retrouvées dans les pages du `man`.

Exercice 1.2.1: Que fait `cut` ?
Cet exercice est une première prise en main de la commande `man`. Vous allez ainsi découvrir une commande qui vous est a priori inconnue. Prenons l'exemple avec la commande `cut`.

1. À l'aide de la commande `man`, indiquer ce que fait la commande `cut` aux fichiers texte qui lui sont passés en arguments.
2. Doit-on passer au moins une option (courte ou longue) en argument à la commande `cut` ?
3. Peut-on passer plus d'une option (courte ou longue) en argument à la commande `cut` ?
4. Doit-on passer au moins un nom de fichier en argument à la commande `cut` ?
5. Peut-on passer plus d'un fichier en argument à la commande `cut` ?
6. Indiquer ce que fait l'option `--version`.

Solution page 37

4 Autres outils pour consulter de l'aide

Le `man` est l'outil le plus utilisé pour trouver de l'aide mais ce n'est pas le seul. Des outils tels que `info` ou `help` proposent également de la documentation sur certaines commandes. Ils n'ont pas rencontré le même succès que `man` mais perdurent encore dans certaines distributions Linux.

4.1 L'aide aux commandes internes avec la commande `help`

La commande `help` propose de la documentation sur les commandes internes au Bash. La liste des commandes internes peut d'ailleurs être obtenue via un appel à la commande `help` sans argument. Les documentations des commandes internes sont également accessibles via le `man` mais selon le système d'exploitation, elles peuvent être plus ou moins concises que via la commande `help`. Notons par exemple que la commande `cd` n'est pas documentée dans le `man` mais elle l'est dans `help`.

```
$ help cd
cd: cd [-L|-P] [dir]
    Change the current directory to DIR.  The variable $HOME is the
    default DIR.  The variable CDPATH defines the search path for
    the directory containing DIR.  Alternative directory names in
    CDPATH
```

Enfin, il est à signaler que la commande **man bash** vous donne la documentation complète sur la syntaxe utilisée pour une ligne de commande. Bien sûr, vous y retrouverez la documentation sur toutes les commandes internes.

4.2 L'aide des projets GNU avec la commande info

Historiquement, l'utilitaire **info** était un concurrent de **man**. La documentation des commandes était rédigée via le format texinfo et visualisée via le mode info de l'éditeur de texte Emacs. Texinfo est resté le format de documentation officiel du projet GNU. L'outil **info** permet d'accéder à la documentation des commandes et bibliothèques GNU (y compris le Bash) ainsi qu'aux pages du **man**. Son principal avantage réside dans la possibilité de naviguer dans la documentation en hypertexte.

La commande **info** représente la documentation sous forme de structure arborescente en utilisant des commandes simples pour parcourir l'arborescence et suivre les références croisées. Par exemple, la navigation s'effectue par les commandes suivantes :

- n va à la page suivante.
- p va à la page précédente.
- u va au nœud parent.
- l va au dernier nœud visité.

Pour suivre une référence croisée, il suffit de déplacer le curseur sur un lien (mot précédé du caractère '*') et d'appuyer sur la touche de retour à la ligne. Vous pouvez aussi accéder directement à une section en la spécifiant en argument. Par exemple **info bash**.

Un appel à la commande **info** sans argument démarre la commande à la racine de la documentation du projet GNU. La commande H affiche les commandes internes à **info**. La commande d indique de revenir à la racine de la documentation et la commande q arrête la commande **info**.

5 Trouver une commande avec la commande apropos

apropos est une commande qui permet de lister les manuels dont la description comprend les mots passés en arguments. Les manuels sont indexés dans une base qui comporte le nom de la commande et une courte description. **apropos** recherche un mot clé passé en argument dans ces deux parties puis affiche les commandes correspondantes. L'affichage de la description courte d'une commande s'obtient par la commande **whatis** :

```
$ whatis apropos
apropos (1)          - Chercher le nom et la description des pages du manuel
```

Il peut arriver que vous ayez besoin d'une commande qui effectue une certaine action mais vous n'en connaissez pas le nom. La commande **apropos** vous sera alors d'une grande aide. Imaginons par exemple que vous ayez besoin d'un convertisseur d'encodage de fichier mais vous en ignorez le nom. Le mot clé **encoding** permettra à **apropos** de trouver l'utilitaire cherché :

```
$ apropos encoding
bind_textdomain_codeset (3) - set encoding of message translations
```

```
chardet (1)          - universal character encoding detector
charmap (5)          - character symbols to define character encodings
Encode::Locale (3pm) - Determine the locale encoding
iconv (1)            - Convert encoding of given files from one encoding
                     to another
manconv (1)          - convert manual page from one encoding to another
preconv (1)          - convert encoding of input files to something GNU
                     troff understands
utf-8 (7)            - an ASCII compatible multibyte Unicode encoding
utf8 (7)             - an ASCII compatible multibyte Unicode encoding
```

Un rapide coup d'œil sur la description des commandes réduira le seul candidat à `iconv`, commande qui convertit les encodages des fichiers.

La puissance de cet outil était particulièrement appréciée du temps où les moteurs de recherches sur Internet étaient peu accessibles. Aujourd'hui encore, `apropos` permet un gain de temps non négligeable, notamment sur les machines qui ne disposent pas d'interface graphique ou d'accès à Internet.

A noter que dans le contexte de la Weblinux, le nombre de commandes proposées est limité. Le résultat de la commande `apropos` est par conséquent plus pauvre. L'exemple ci-dessus ne provient pas de la Weblinux.

✍ **Challenge C12Q4**

À propos de la commande `apropos`

Exercice 1.2.2: Trouver les bons outils
Utiliser la commande `apropos` pour trouver les commandes permettant d'effectuer les tâches suivantes :

- **copier** un fichier ou un répertoire,
- **trouver** un fichier ou un répertoire,
- **effacer** un fichier.

Solution page 37

6 Conclusion

Dans cette activité nous avons vu que votre système d'exploitation vous propose toute l'aide nécessaire à l'utilisation des commandes. Il n'est pas nécessaire de faire des recherches sur Internet pour utiliser votre shell Bash. Nous avons vu l'aide qui est intégrée directement dans les commandes et qui est affichée à travers l'option `-h` ou lors d'une erreur de syntaxe. Nous avons également vu l'aide décrite dans des manuels. Ceux-ci peuvent être en français et consultés au moyen de la commande man. Les manuels sont aussi utilisés par la commande `apropos` qui permet de trouver les commandes associées à une action à effectuer. Grâce à cette activité, vous n'aurez plus droit à RTFM [2] en réponse aux questions que vous poserez sur les forums spécialisés.

2. RTFM : Read The Fucking Manual.

Solutions des exercices

Solution de l'exercice 1.2.1 page 34:

1. Grâce à la partie NOM du man de la commande `cut`, nous apprenons qu'elle sert à « Supprimer une partie de chaque ligne d'un fichier ».
2. La partie SYNOPSIS du man de la commande `cut` contient le texte ci-dessous :

```
SYNOPSIS
       cut [OPTION] ... [FICHIER] ...
```

   La notation avec des crochets (`'[]'`) autour de OPTION indique que l'argument noté OPTION est optionnel. A priori, il n'est donc pas nécessaire de passer une option en argument de la commande `cut`.
3. La présence des trois petits points (...) après [OPTION] indique qu'on peut en spécifier plusieurs.
4. La présence des crochets (`'[ ]'`) autour de FICHIER indique que le fichier est optionnel. A priori il n'est donc pas nécessaire de passer un nom de fichier en argument à la commande `cut`.
5. La présence des trois petits points ((`'...'`) après [FICHIER] indique qu'on peut en spécifier plusieurs.
6. La partie DESCRIPTION du man de la commande `cut` contient le texte ci-dessous :

```
DESCRIPTION
       Afficher  des  parties  selectionnees des lignes de chaque
   FICHIER vers la sortie standard.
       Les parametres obligatoires pour les options de forme longue le
   sont aussi pour les options de forme courte.
       (...)
        --version
              Afficher le nom et la version du logiciel et quitter
```

   On y apprend la signification de l'option longue `--version` qui est « Afficher le nom et la version du logiciel (cut) et quitter ».

Solution de l'exercice 1.2.2 page 36:

1. L'appel à `apropos` avec l'argument `copier` nous apprend que la commande `cp` répond à nos besoins :

```
$ apropos copier
cp (1)               - Copier des fichiers et des repertoires
```

2. L'appel à `apropos` avec l'argument `chercher` nous indique que la commande `find` répond à nos besoins :

```
$ apropos chercher
apropos (1)          - Chercher le nom et la description des pages de
   manuel
find (1)             - Rechercher des fichiers dans une hierarchie de
   repertoires
```

3. L'appel à `apropos` avec l'argument `effacer` nous apprend que la commande `rm` répond à nos besoins :

```
$ apropos effacer

rm (1)               - Effacer des fichiers et des repertoires
clear (1)            - Effacer la console
clear_console (1) [clear] - Effacer la console
```

Notons que les mots clefs à utiliser sont souvent en anglais et qu'il faut parfois essayer plusieurs synonymes du mot clef (*e.g.* effacer, supprimer) avant que `apropos` indique la commande correspondante.

Activité 1.3

Système de fichiers et répertoires

1 Introduction

Cette activité porte sur l'organisation des fichiers du système d'exploitation, l'objectif est de vous rendre efficace dans la consultation et la manipulation des fichiers stockés.

Sous Unix, toutes les ressources de votre ordinateur sont représentées par des fichiers. Par exemple, les communications, les périphériques et les données passent par le système de fichiers.
Le système de fichiers est la partie la plus visible d'un système d'exploitation. Il se charge de gérer le stockage et la manipulation de fichiers (lecture et écriture). Les périphériques de stockage habituellement utilisés (disques durs, disquettes, clefs USB ou disques optiques) se présentent au système d'exploitation sous la forme d'une succession de blocs de données numérotés. Cette organisation basique ne définit aucune notion de fichier ou de répertoire, et permet seulement de stocker des données les unes à la suite des autres.

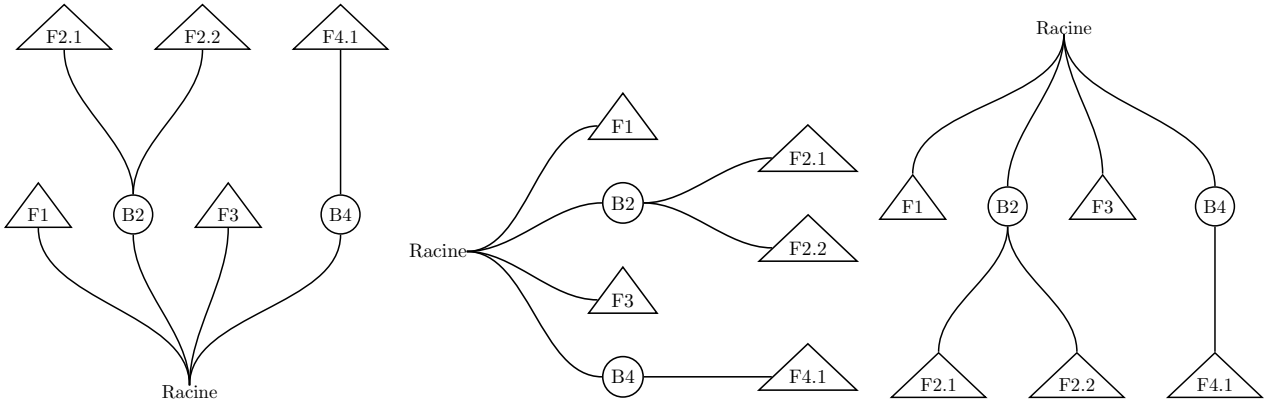
Le système d'exploitation présente les données à l'utilisateur et aux programmes selon une organisation structurée, sous la forme de répertoires et de fichiers. Pour pouvoir stocker ces données structurées sur un périphérique, il faut utiliser un format qui les représente sous la forme d'une succession de blocs de données : c'est ce qu'on appelle un **système (de gestion) de fichiers**.

Au cours du temps, les systèmes de fichiers Unix se sont modernisés pour augmenter la capacité de stockage, accélérer les accès, être journalisés (il existe un journal des dernières transactions), gérer finement les droits de chaque utilisateur, virtualiser les accès (à travers le réseau, modifier le support réel, ...) ou encore permettre le chiffrement.

2 Arborescence

La structure de stockage des fichiers est organisée hiérarchiquement selon une arborescence. Une arborescence est un graphe en forme d'arbre composé d'éléments pour lesquels les feuilles aux extrémités des branches sont les fichiers et les branches sont des répertoires. Une branche peut avoir plusieurs feuilles mais une feuille n'est attachée qu'à une seule branche. Une branche en plus des feuilles peut avoir plusieurs sous-branches.

L'arborescence démarre d'un répertoire particulier nommé racine ou `root` en anglais. La figure 1.3.1a représente un arbre dans lequel les nœuds dits « branches » sont représentés par des nœuds circulaires et les feuilles sont représentées par des nœuds en triangles. L'organisation en arborescence offre à l'utilisateur une vision hiérarchique et un moyen de naviguer dans les répertoires du système de fichiers. En effet, depuis la racine il existe un chemin unique vers chaque élément de l'arborescence.



(a) Orienté vers le haut (b) Orienté vers la droite (c) Orienté vers le bas

FIGURE 1.3.1 – Représentation d'un arbre.

Notons que cette analogie ne correspond pas aux représentations habituelles dans lesquelles l'arbre est renversé (voir figure 1.3.1c) ou orienté vers la droite (voir figure 1.3.1b).

Une organisation arborescente introduit les notions de parent et de fils d'un nœud. Un nœud P est le parent d'un nœud A si il existe un lien direct entre P et A et que P est sur le chemin entre la racine et A. Autrement dit, le parent du nœud A est le répertoire dans lequel il se situe. Par exemple, dans la figure 1.3.1, $B2$ est le parent de $F2.2$ mais aussi de $F2.1$. La racine est le parent de $B2$. De manière similaire on dira que $B2$ est un fils de la racine et que $F2.1$ et $F2.2$ sont les fils de $B2$.

2.1 Présentation du système de fichiers

Avec les systèmes de type Unix, le système de fichiers démarre à la racine, à la base de l'arborescence. Il n'y a qu'une seule racine pour l'ensemble du système de fichiers. La racine est notée par le caractère `'/'`. Elle précède la description des chemins d'accès (*pathname*).

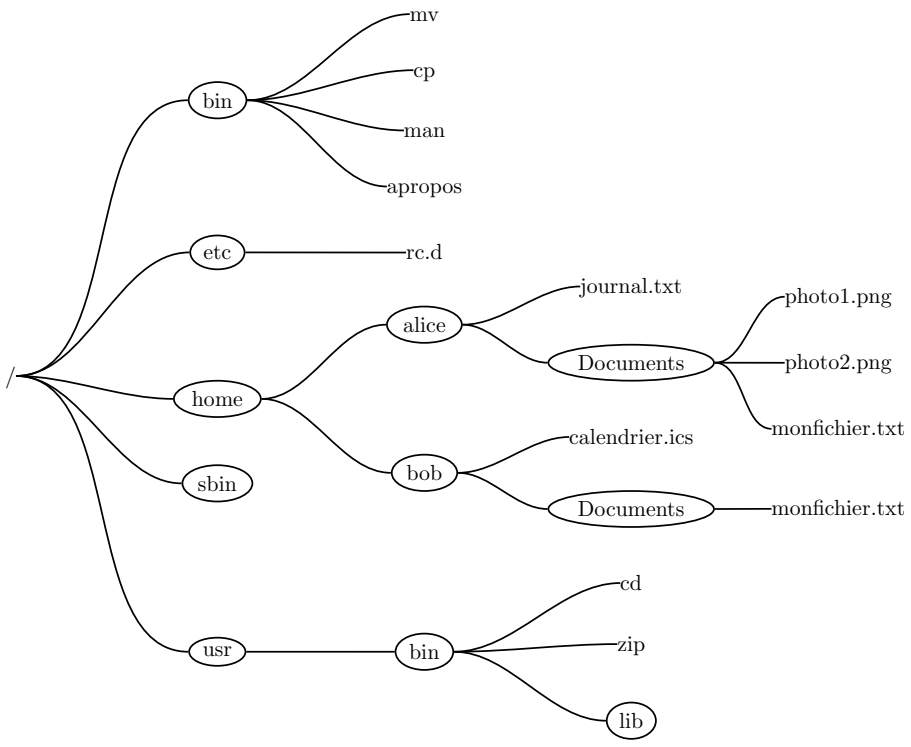


FIGURE 1.3.2 – Arborescence du système de fichiers.

Une arborescence type d'un système de gestion de fichiers ressemble à celle de la figure 1.3.2. Sous la racine notée `/`, nous allons trouver plusieurs répertoires :

- des répertoires systèmes comme `bin` ou `usr` qui contiennent les commandes que vous allez utiliser ;
- des répertoires importants au bon fonctionnement de votre système d'exploitation comme `etc` car il contient les fichiers de configuration de votre système d'exploitation ;
- des répertoires spécifiques comme `tmp` qui sert de stockage temporaire (au redémarrage de la machine, son contenu est supprimé) ;
- le répertoire des utilisateurs `home` qui comporte le répertoire personnel de chaque utilisateur de la machine, aussi appelé *home directory* en anglais. Ainsi l'utilisateur *alice* possède-t-il son propre répertoire appelé `alice` et situé dans `/home`.

3 Chemins d'accès dans l'arborescence

Le chemin d'accès est l'expression pour localiser un fichier ou un répertoire dans l'arborescence. Il est constitué d'une liste de noms identifiant les branches du chemin. Il existe deux types de chemins. Le chemin dit absolu a l'avantage de désigner un élément de l'arborescence sans ambiguïté mais il peut être long à écrire. Un chemin relatif est en général beaucoup plus court à écrire mais il peut y avoir une ambiguïté sur sa signification, c'est-à-dire sur le noeud de l'arbre auquel il fait référence.

Relatif vs absolu

Tout chemin qui commence par `/` est un chemin absolu. À l'inverse, les autres chemins sont relatifs. Un chemin relatif a comme point de départ le répertoire courant.

Chemin absolu

Un chemin absolu (d'accès) sur le système de fichiers permet d'identifier sans ambiguïté un élément de l'arborescence, qu'il s'agisse d'un répertoire ou d'un fichier. Pour construire un chemin absolu, on commence par l'élément ciblé que l'on préfixe par `/` puis on y ajoute le nom du répertoire parent, c'est-à-dire la branche qui le relie vers la racine. On réitère l'opération jusqu'à atteindre la racine. Par exemple, la construction du chemin du fichier `photo1.png` de la figure 1.3.2 est le suivant :

- Nom du fichier : `photo1.png`
- première étape : `Documents/photo1.png`,
- deuxième étape : `alice/Documents/photo1.png`,
- troisième étape : `home/alice/Documents/photo1.png`
- dernière étape : `/home/alice/Documents/photo1.png`.

Le chemin d'accès `/home/alice/Documents/photo1.png` est dit absolu car il identifie sans ambiguïté le fichier `photo1.png` dans l'arborescence. Tout chemin qui commence par la racine est un chemin absolu.

Chemin relatif

Les chemins qui ne commencent pas par la racine sont dit relatifs. Leur signification dépend d'un répertoire de départ appelé répertoire courant. Lorsque l'utilisateur ouvre une session, le répertoire courant est son répertoire personnel. Au cours de la session, il peut utiliser la commande `cd` pour changer de répertoire courant.

Les chemins relatifs peuvent être ambigus. Prenons par exemple le cas du système de fichier proposé par la figure 1.3.2 dans lequel il peut y avoir deux fichiers qui correspondent au chemin relatif `Documents/monfichier.txt`. Si le répertoire courant est le répertoire personnel de *alice*, alors il

s'agira de `/home/alice/Documents/monfichier.txt` tandis qu'il s'agira de `/home/bob/Documents/monfichier.txt` si le répertoire courant est le répertoire personnel de *bob*.

3.1 Symboles pour le chemin d'accès

Le nom d'un répertoire ou d'un fichier ne doit pas comporter plus de 255 caractères. Il est de bonne pratique de n'utiliser que des caractères alphabétiques (sans accent) et des chiffres. Il faut préférer utiliser le caractère souligner `'_'` à l'espace comme séparateur de mot dans un nom. L'espace est à proscrire car elle a une fonction spéciale dans la ligne de commande. Si il y a un espace dans le nom, il faut précéder ce caractère par le caractère anti-slash '\'. À savoir aussi, les minuscules et les majuscules sont distinguées.

En plus du séparateur `'/'` et des noms, un chemin peut contenir des symboles qui ont des significations spéciales. Un nom réduit au caractère `'.'` fait référence au répertoire courant. Le symbole `'..'` fait référence au répertoire parent du répertoire courant. Le caractère `'~'` (appelé tilde), fait référence au répertoire personnel de l'utilisateur. Nous y reviendrons dans l'activité 2.2. Le tableau ci-dessous synthétise les symboles spéciaux utilisés dans le chemin d'accès.

`..`	répertoire parent
`/`	sépare les branches dans un chemin.
`.`	répertoire courant.
`~`	répertoire personnel de l'utilisateur

Si le répertoire courant est le répertoire `Documents` de l'utilisateur *alice*, alors les chemins relatifs suivants font tous référence au même fichier désigné par le chemin absolu `/home/alice/Documents/monfichier.txt` :

```
monfichier.txt
./monfichier.txt
../Documents/monfichier.txt
~/Documents/monfichier.txt
```

Cela n'est pas utilisé en pratique mais notez que vous pouvez utiliser les symboles spéciaux `'.'` et `'..'` dans un chemin absolu. Par exemple, les chemins suivants sont également des chemins valides pour indiquer le fichier `/home/alice/Documents/monfichier.txt` :

```
/home/bob/Documents/../../alice/Documents/monfichier.txt
/home/alice/./././././././././../Documents/monfichier.txt
```

4 Les commandes de navigation

Avant de réaliser toute action, il est nécessaire de savoir où l'on se situe, c'est-à-dire savoir quel est le répertoire courant. La commande `pwd` (*Print Working Directory*) affiche le chemin d'accès absolu du répertoire courant. Elle indique la position dans l'arborescence :

```
$ pwd
/home/alice
```

Cette commande est informative. Elle ne modifie pas le répertoire courant, ni le système de fichiers.

Pour changer de répertoire courant et donc se déplacer dans le système de fichiers, vous utiliserez la commande `cd` (*Change Directory*). La commande `cd` change le répertoire courant par le répertoire dont le chemin est passé en argument, qu'il soit relatif ou absolu :

```
$ pwd
/home
$ cd alice/Compagnon
$ pwd
/home/alice/Compagnon
$ cd /
$ pwd
/
$ cd /home/alice/Compagnon/
$ pwd
/home/alice/Compagnon
```

Lorsqu'elle est appelée sans argument, la commande `cd` replace le répertoire courant sur le répertoire personnel de l'utilisateur. Vous pouvez utiliser les caractères spéciaux du chemin d'accès.

```
$ pwd
/etc
$ cd
$ pwd
/home/alice
$ cd ..
$ pwd
/home
$ cd ~
$ pwd
/home/alice
```

Maintenant que l'on sait se déplacer dans le système de fichiers, il faut pouvoir observer son contenu. C'est le rôle de la commande `ls` (*List Segments*).

```
$ pwd
/home/alice
$ ls /home/bob
calendrier.ics
Documents
```

Sans argument, la commande `ls` liste les noms des éléments contenus dans le répertoire courant :

```
$ pwd
/home/alice/Compagnon/A13
$ ls
Documents Journal.txt
```

Notez que `ls` comprend de nombreuses options. Il convient de consulter le manuel de la commande `ls` pour aller plus loin sur son usage. Par exemple, l'option `-l` (*long*) retourne des informations détaillées sur le contenu du répertoire :

```
$ ls -l
-rw-r--r-- alice user 123 Mar 13 16:40 journal.txt
drwxr-xr-x alice user   6 Mar 16 16:41 Documents
```

On apprend que `Documents` est un répertoire (la ligne est préfixée par un `d`) modifié le 16 mars à 16h41 et que `journal.txt` est un fichier de 123 octets modifié le 13 mars à 16h41.

Exercice 1.3.1: Chemins absolus et relatifs

En se basant sur l'arborescence définie dans la figure 1.3.2, indiquer si les répertoires et fichiers ci-dessous font référence à ceux de Bob, d'Alice ou s'ils n'existent pas :

1. `/home/bob/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire racine.
2. `/home/bob/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire personnel de Bob.
3. `/home/bob/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire d'Alice.
4. `./Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire personnel de Bob.
5. `./Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire d'Alice.
6. `../bob/../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire personnel de Bob.
7. `../bob/../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire personnel de Alice.
8. `../bob/../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire `/sbin`
9. `/home/bob/Documents/../../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire racine.
10. `/home/bob/Documents/../../alice/./././../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire racine.
11. `/home/bob/Documents/../../alice/./././../../alice/Documents/monfichier.txt` lorsque qu'on est placé dans le répertoire racine.

Solution page 49

✍ **Challenge C13Q1**

Commandes de navigation dans l'arborescence

✍ **Challenge C13Q2**

Chemin absolu et utilisation du caractère tilde

5 Les commandes de modification de l'arborescence

Les commandes du shell Bash nous permettent de créer ou supprimer des fichiers et des répertoires, de les copier et de les déplacer.

5.1 Création et suppression de répertoires : `mkdir` et `rmdir`

La commande `mkdir` (*MaKe DIRectory*) crée un répertoire et va servir à organiser votre arborescence pour y ranger vos fichiers.

```
$ mkdir monRepertoire
$ ls -l
```

```
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
-rw-r--r-- alice user 0 Avril 14 16:40 fichier1.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
```

Il est possible de créer plusieurs répertoires en même temps :

```
$ mkdir monRepertoire1 monRepertoire2
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire2
-rw-r--r-- alice user 0 Avril 14 16:40 fichier1.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
```

La suppression du répertoire se fait avec la commande **`rmdir`** (*ReMove DIRectory*) suivie du nom du (ou des) répertoire(s) à supprimer. Cette commande fonctionne si le ou les répertoires à supprimer sont vides. Pour supprimer un répertoire non vide, il faut utiliser la commande **`rm`** présentée dans le paragraphe suivant.

```
$ rmdir monRepertoire2
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 0 Avril 14 16:40 fichier1.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
```

5.2 Création et suppression de fichiers quelconques : `touch` et `rm`

La commande **`touch`** permet de créer un fichier vide :

```
$ cd ~
$ ls
fichier1.txt  Documents
$ touch monfichier.txt
$ ls  -l
total 3
drwxr-xr-x    1 alice    user             0 Jan 26 15:08 Documents
-rw-r--r--    1 alice    user          3024 Jan 26 15:08 fichier1.txt
-rw-r--r--    1 alice    user             0 Jan 26 15:38 monfichier.txt
```

Nous verrons dans l'activité 1.5 comment ajouter du contenu dans les fichiers que nous créons avec la commande **`touch`**.

✍ **Challenge C13Q3**

Création de fichiers et répertoires

La suppression de fichier se fait par la commande **`rm`** (*ReMove*) suivie du nom (chemin) du fichier à supprimer :

```
$ cd ~
$ ls
fichier1.txt  monfichier.txt Documents
```

```
$ rm monfichier.txt
$ ls
fichier1.txt  Documents
```

Convention d'ordre des arguments

Lors de la copie ou du déplacement, à retenir : *ancien nouveau*. Ce qui signifie que le premier argument désigne l'ancien élément, le second désigne le nouveau élément.

Notez que la commande `rm` peut également supprimer des répertoires : l'option `-r` permet de supprimer récursivement l'arborescence contenue dans le répertoire. Utilisez également l'option `-f` pour ne pas avoir à confirmer la suppression de chaque fichier contenu dans la sous-arborescence du répertoire. Ainsi, `rm -rf Documents` permet de supprimer l'intégralité du répertoire `Documents` en une commande. Avant de valider une telle commande, il est conseillé de relire celle-ci au moins deux fois.

✍ **Challenge C13Q4**

Manipulation de chemins relatifs

5.3 Déplacer et copier : `mv` et `cp`

Vous pouvez facilement renommer, déplacer mais également copier un fichier ou un répertoire. À l'aide de la commande `mv`, il est possible de renommer le fichier `fichier1.txt`. Les arguments de cette commande suivent la convention d'ordre des systèmes de type Unix, à savoir source/ancien nom suivi de la destination/nouveau nom.

```
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 0 Avril 14 16:40 fichier1.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
$ mv fichier1.txt fichier123.txt
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 0 Avril 14 16:40 fichier123.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
```

Si le nom du fichier de destination existe déjà, celui-ci sera écrasé par le fichier source. Il faut être très attentif lors de l'utilisation de la commande `mv` car il est facile de se tromper et ainsi de perdre des informations. Dans l'exemple ci-dessous, Alice va perdre le contenu du fichier `fichier2.txt` :

```
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 0 Avril 14 16:40 fichier1.txt
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
$ mv fichier1.txt fichier2.txt
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 0 Avril 14 16:41 fichier2.txt
```

À l'aide de la commande `mv` (abréviation de *MoVe*), il est également possible de déplacer le fichier `fichier1.txt` vers le répertoire `monRepertoire` :

```
$ mv fichier1.txt monRepertoire
$ ls -l monRepertoire
-rw-r--r-- alice user 0 Avril 14 16:44 fichier1.txt
```

Notez qu'il en va de même pour le déplacement ou renommage de répertoires :

```
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
$ mv monRepertoire monRepertoire1
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
$ ls -l monRepertoire1
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
```

Notez que si le nom du répertoire de destination n'existe pas, *monRepertoire* ne sera pas déplacé mais simplement renommé :

```
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
$ mv monRepertoire monRepertoire2
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire2
```

La commande `cp` (abréviation de *CoPy*) permet de copier des fichiers :

```
$ cp fichier1.txt fichier-bis.txt
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
-rw-r--r-- alice user 463 Avril 14 16:41 fichier1.txt
-rw-r--r-- alice user 463 Sept 5 12:01 fichier1-bis.txt
```

Notez que la taille du fichier copié est identique. Cependant l'heure de création a changé car c'est un nouveau fichier qui a été créé. Ce n'était pas le cas avec la commande `mv`.

Si vous souhaitez appliquer la commande `cp` à un répertoire, il faudra ajouter l'option `-R` pour spécifier qu'il faut copier récursivement l'intégralité de la sous-arborescence du répertoire :

```
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
$ cp -R monRepertoire monRepertoire-bis
$ ls -l
drwxr-xr-x alice user 0 Avril 14 16:42 monRepertoire
drwxr-xr-x alice user 0 Sept 5 12:02 monRepertoire-bis
drwxr-xr-x alice user 0 Avril 14 16:43 monRepertoire1
```

✍ **Challenge C13Q5**

Copie et renommage

✍ **Challenge C13Q6**

Suppression de fichiers

Exercice 1.3.2: Copier, déplacer et renommer
On considère l'arborescence définie à la figure 1.3.2. L'utilisateur a placé le répertoire courant dans le répertoire personnel d'Alice. Pour chacune des commandes ci-dessous, indiquer les fichiers qui existent encore, ceux qui ont été dupliqués ou renommés. Donner le ou les chemins (s'ils ont été copiés) permettant d'accéder à leur contenu.

1. `mv journal.txt journal2.txt`
2. `mv journal.txt /home/bob`
3. `mv Documents/photo1.png .`
4. `mv Documents/photo1.png ./Documents/photo3.png`
5. `cp Documents/photo1.png ./Documents/photo3.png`
6. `cp Documents/photo1.png ./Documents/photo2.png`
7. `mv Documents/photo1.png ./Documents/photo2.png`

Solution page 49

6 Conclusion

Dans cette activité, nous avons présenté l'arborescence du système de fichiers. Nous avons identifié des répertoires particuliers comme : la racine, le répertoire personnel, le répertoire courant.

L'accès à un fichier ou un répertoire est décrit par un chemin d'accès. Celui-ci est dit relatif si le point de départ est le répertoire courant, ou absolu si le point de départ est la racine.

Vous avez vu les principales commandes pour modifier et naviguer dans l'arborescence du système de fichiers. Cette compétence est indispensable à la poursuite du MOOC.

Vous trouverez ci-dessous un tableau pour vous rappeler les commandes indispensables :

`cd`	changer le répertoire courant
`mv`	renommer ou déplacer.
`cp`	copier.
`touch`	créer un fichier.
`rm`	supprimer.
`mkdir`	créer un répertoire.

Solutions des exercices

Solution de l'exercice 1.3.1 page 44:

1. `/home/bob/Documents/monfichier.txt` est un chemin absolu. Sa signification ne dépend pas de l'endroit où on se situe dans l'arborescence. Il fait donc référence au fichier de Bob.
2. `/home/bob/Documents/monfichier.txt` est un chemin absolu. Sa signification ne dépend pas de l'endroit où on se situe dans l'arborescence. Il fait donc référence au fichier de Bob.
3. `/home/bob/Documents/monfichier.txt` est un chemin absolu. Sa signification ne dépend pas de l'endroit où on se situe dans l'arborescence. Il fait donc référence au fichier de Bob.
4. `./Documents/monfichier.txt` est un chemin relatif. Sa signification dépend de l'endroit où on se situe dans l'arborescence. Si on est dans le répertoire personnel de Bob, il est équivalent au chemin absolu `/home/bob/Documents/monfichier.txt` et fait donc référence à celui de Bob.
5. `./Documents/monfichier.txt` est un chemin relatif. Sa signification dépend de l'endroit où on se situe dans l'arborescence. Si on est dans le répertoire personnel de Alice, il est équivalent au chemin absolu `/home/alice/Documents/monfichier.txt` et fait donc référence à celui de Alice.
6. `../bob/../alice/Documents/monfichier.txt` est un chemin relatif. Cependant, il commence par `../` qui, lorsqu'on est situé dans le répertoire personnel d'un utilisateur, fait référence à `/home`. Il n'y a donc plus d'ambiguïté sur le répertoire auquel il fait référence tant qu'on est dans le répertoire personnel d'un utilisateur (*e.g.* `/home/alice` ou `/home/bob`). Il fait référence à `/home/bob/../alice/Documents/monfichier.txt` c'est-à-dire à `/home/alice/Documents/monfichier.txt` et donc à celui d'Alice.
7. `../bob/../alice/Documents/monfichier.txt`, lorsque qu'on est placé dans le répertoire personnel de Alice, fait référence au fichier d'Alice, pour la même raison que ci-dessus.
8. `../bob/../alice/Documents/monfichier.txt`, lorsque qu'on est placé dans le répertoire `/sbin`, `../` fait référence à `/`. Ainsi, `../bob` fait référence à `/bob`, un répertoire qui n'existe pas.
9. `/home/bob/Documents/../../alice/Documents/monfichier.txt` est un chemin absolu. Il fait référence au fichier d'Alice.
10. `/home/bob/Documents/../../alice/./././../alice/Documents/monfichier.txt` est un chemin absolu. Il fait référence au fichier d'Alice.
11. `/home/bob/Documents/../../alice/./././../../alice/Documents/monfichier.txt` est un chemin absolu. `/home/bob/Documents/../../alice/./././../../alice` est équivalent à `/alice` or, ce répertoire n'existe pas.

Solution de l'exercice 1.3.2 page 48:

1. `mv journal.txt journal2.txt` Le contenu de `journal.txt` a été renommé en `journal2.txt`. Il est toujours dans le même répertoire et son chemin absolu est `/home/alice/journal2.txt`
2. `mv journal.txt /home/bob` Le contenu de `journal.txt` a été déplacé dans `/home/bob`. Son chemin absolu est `/home/bob/journal.txt`

3. `mv Documents/photo1.png .` Le contenu de `/home/alice/Documents/photo1.png` a été déplacé dans le répertoire courant à savoir `/home/alice`. Son chemin absolu est `/home/alice/photo1.png`
4. `mv Documents/photo1.png ./Documents/photo3.png` Le fichier `photo1.png` a été renommé en `photo3.png`. Il est toujours dans le même répertoire et son chemin absolu est `/home/alice/Documents/photo3.png`
5. `cp Documents/photo1.png ./Documents/photo3.png` Le fichier `photo1.png` a été copié dans un fichier de nom `photo3.png`. Leurs chemins absolus sont `/home/alice/Documents/photo1.png` et `/home/alice/Documents/photo3.png`
6. `cp Documents/photo1.png ./Documents/photo2.png` Le fichier `photo1.png` a été copié dans un fichier de nom `photo2.png` et a écrasé le contenu de ce dernier. Leurs chemins absolus sont `/home/alice/Documents/photo1.png` et `/home/alice/Documents/photo2.png`. L'ancien contenu de `/home/alice/Documents/photo2.png` est désormais inaccessible.
7. `mv Documents/photo1.png ./Documents/photo2.png` Le fichier `photo1.png` a été renommé en `photo2.png`. Il est toujours dans le même répertoire et son chemin absolu est `/home/alice/Documents/photo2.png`. L'ancien contenu de `/home/alice/Documents/photo2.png` est désormais inaccessible.

Activité 1.4

Les utilisateurs et leurs droits

1 Introduction

Un système d'exploitation de type Unix est un système d'exploitation multi-tâches et multi-utilisateurs. Cela signifie que sur une même machine plusieurs personnes peuvent travailler simultanément.

Le système doit donc pouvoir gérer plusieurs utilisateurs en même temps en assurant à la fois le partage des ressources (espace disque, utilisation de la mémoire, périphériques, ...), la confidentialité des données de chaque utilisateur et bien sûr l'intégrité de l'arborescence des répertoires et des fichiers.

Puisque plusieurs personnes peuvent être connectées en même temps, le système doit pouvoir identifier clairement chacun des utilisateurs ainsi que les ressources auxquelles ils ont accès et plus généralement qui a le droit de faire quoi.

Ainsi, chaque personne autorisée à utiliser un système de type Unix se voit attribuer un compte utilisateur et il existe un ensemble de règles qui régissent ce qu'elle a le droit de faire. Nous allons découvrir dans cette activité les mécanismes mis en œuvre pour gérer les utilisateurs.

2 Identification des utilisateurs

Il y a deux types d'utilisateurs, comme le montre la figure 1.4.1 :

- Un super-utilisateur qui a le droit de faire tout ce qu'il veut sur le système, absolument tout : créer des utilisateurs, leur accorder des droits, supprimer des utilisateurs, avoir accès à leurs données, modifier le système. Ce super-utilisateur s'appelle *root*. Cet utilisateur, c'est l'administrateur du système.
- Et les autres utilisateurs. Ceux-là n'ont qu'une possibilité d'action limitée et surtout pas la possibilité de modifier le système. Ces utilisateurs peuvent être répartis dans différents groupes.

2.1 Utilisateurs et groupes

Pour être identifié sur un système de type Unix, il faut posséder un compte utilisateur, créé par le super-utilisateur, et caractérisé par un identifiant de compte et un mot de passe.

Pour se connecter sur le système par une console et ouvrir une session de travail, il faut entrer son identifiant à l'invite `login` puis son mot de passe à l'invite `passwd`. Afin de rester confidentiel, la saisie du mot de passe se fait en aveugle.

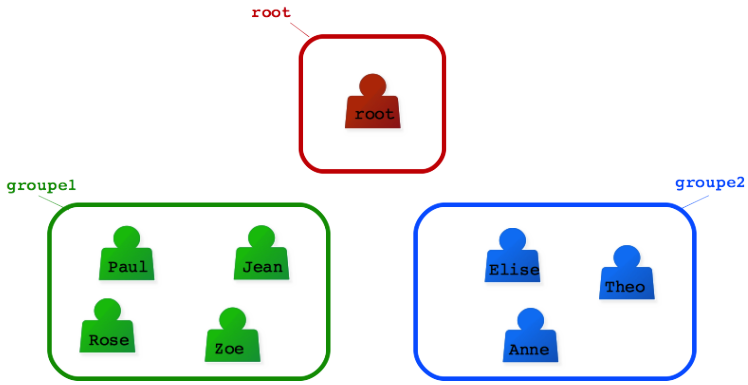


FIGURE 1.4.1 – Utilisateurs et groupes.

En plus de son identifiant de compte, chaque utilisateur est identifié par un numéro unique `uid` (*user identifier*) et appartient à un groupe principal `gid` (*group identifier*) et éventuellement à des groupes secondaires d'utilisateurs.

Le groupe principal est utilisé par le système en relation avec les droits d'accès aux fichiers, nous détaillons cela dans les sections 3 et 4. Chaque utilisateur doit appartenir à un groupe principal.

Les groupes secondaires sont les autres groupes auxquels un utilisateur appartient. Un utilisateur peut au maximum appartenir à 1024 groupes secondaires.

Pour connaître son `uid` et les groupes auxquels on appartient, on peut utiliser la commande `id` (*IDentity*) :

```
$ id
uid=1000(john) gid=1000(python) groups=1000(python),4(adm),
24(cdrom),27(sudo),118(lpadmin)
```

Dans cet exemple, l'identifiant de compte de l'utilisateur porte le nom *john*, son numéro d'identification est 1000, son groupe principal est le groupe *python* identifié par le numéro 1000. On voit aussi qu'il appartient à plusieurs groupes secondaires de `gid` 4, 24, 27 et 118.

✍ **Challenge C14Q1**

Dans quels groupes suis-je ?

2.2 Le fichier `/etc/passwd`

C'est dans ce fichier que se trouvent les informations de connexion de tous les utilisateurs du système. Il s'agit d'un fichier texte où chaque ligne correspond à un utilisateur. Cette ligne est composée de sept champs séparés par le caractère `':'` comme le montre la figure 1.4.2.

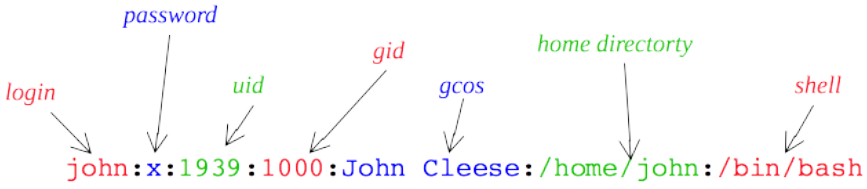


FIGURE 1.4.2 – Format du fichier `/etc/passwd`.

Les informations présentes sur cette ligne sont, dans l'ordre :

— l'identifiant de compte que communément on appelle aussi le nom de l'utilisateur ;
— le mot de passe crypté pour cet utilisateur (sur certain systèmes ce champ ne contient pas le mot de passe crypté mais le caractère `'x'` qui indique que le mot de passe est stocké ailleurs dans un fichier `/etc/shadow` visible uniquement par le super-utilisateur ; si ce champ est vide cela signifie que l'utilisateur n'a pas de mot de passe) ;
— l'`uid` de l'utilisateur ;
— le `gid` de son groupe principal ;
— un champ informatif, appelé *GCOS*, rempli optionnellement. Il contient généralement la description de l'utilisateur et peut être sous la forme de son nom complet ;
— le répertoire personnel de l'utilisateur (*home directory*). Habituellement le nom de ce répertoire est `/home/`*`login`* avec *`login`* l'identifiant de compte ;
— le shell de connexion de l'utilisateur, c'est-à-dire le shell utilisé par l'utilisateur lorsqu'il ouvre une session. Comme indiqué dans le chapitre 0.3, vous savez qu'il existe d'autres types de shell que celui du Bash.

✍ **Challenge C14Q2**

Description de l'utilisateur

2.3 Le fichier `/etc/shadow`

Ce fichier, s'il existe, contient les mots de passe cryptés de tous les utilisateurs, il n'est visible que par le super-utilisateur. Chaque ligne de ce fichier texte est associée à un utilisateur et contient un certain nombre de champs d'information que le super-utilisateur peut renseigner pour la gestion des comptes : la date à laquelle le mot de passe a été modifié pour la dernière fois, le nombre de jours avant expiration du mot de passe, le nombre de jours restant avant le prochain changement obligatoire du mot de passe, le nombre de jours pendant lesquels le compte reste actif après expiration du mot de passe, la date de désactivation du compte.

Remarque : le système installé sur la Weblinux qui accompagne ce cours n'a pas de fichier `/etc/shadow`, le mot de passe crypté est donc stocké dans le fichier `/etc/passwd`.

2.4 Le fichier `/etc/group`

C'est grâce à ce fichier texte qu'un système de type Unix peut gérer des groupes d'utilisateurs. Chaque ligne correspond à un groupe. Une ligne est composée de champs séparés par le caractère `:` . Ces champs sont :
— le nom du groupe ;
— le mot de passe du groupe (ce champ est rarement utilisé ; par contre s'il est renseigné, un utilisateur du groupe qui veut accéder à une ressource appartenant au groupe devra saisir ce mot de passe) ;
— le numéro unique d'identification du groupe ;
— la liste des utilisateurs qui appartiennent au groupe.

3 Les droits d'accès

On a vu que chaque utilisateur d'un système de type Unix se voit attribuer un identifiant de compte ainsi qu'un répertoire personnel rangé sous le répertoire `/home`. Ce répertoire personnel porte le nom de l'identifiant de compte de l'utilisateur et lui appartient. C'est dans ce répertoire qu'il va pouvoir stocker ses fichiers.

Les fichiers bénéficient d'une protection en lecture, écriture et exécution, c'est-à-dire qu'un utilisateur peut choisir que ses fichiers soient lisibles et/ou modifiables par d'autre utilisateurs et il peut empêcher ou autoriser que d'autres utilisateurs lancent ses propres programmes. C'est le principe des droit d'accès.

3.1 Cas d'un fichier ordinaire

Comme l'illustre la figure 1.4.3, les droits d'accès sont définis suivant trois publics différents :
- le propriétaire du fichier ;
- les membres du groupe propriétaire du fichier ; par défaut à la création du fichier ce groupe est le groupe principal du propriétaire ;
- les autres utilisateurs du système.

Pour chacun de ces publics il existe trois droits d'accès :
- le droit de lecture : symbole `r` (*read*) ;
- le droit d'écriture : symbole `w` (*write*) ;
- le droit d'exécution : symbole `x` (*executable*).

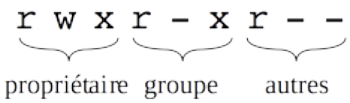


FIGURE 1.4.3 – Droits d'accès.

Pour un fichier ordinaire, le droit de lecture signifie que l'utilisateur peut lire le fichier, ce qui permet entre autre de le copier. Le droit d'écriture signifie que l'utilisateur est autorisé à modifier le contenu et le droit d'exécution permet de considérer le fichier comme une commande. Notons que le droit d'exécution n'a d'intérêt que si le fichier est réellement exécutable, par exemple s'il s'agit d'un script écrit dans un langage connu du système ou d'un fichier binaire issu de la compilation d'un programme.

On peut afficher les droits d'accès d'un fichier avec la commande `ls -l` :

```
$ ls -l poisson.txt
-rwxr-xr--    1 john    python    8 2017-02-12 08:23 poisson.txt
```

Le premier caractère du champ `-rwxr-xr--` indique le type de fichier, ici le symbole `-` signifie qu'il s'agit d'un fichier ordinaire. Dans le cas d'un répertoire, il y aurait la lettre `d`. Les autres symboles `-` présents dans le champ indiquent l'absence de droit.

Les caractères 2 à 4 (`rwx`) indiquent dans l'ordre les droits du propriétaire *john* (en lecture, écriture, exécution), les caractères 5 à 7, les droits des utilisateurs qui appartiennent au groupe *python* (`r-x`) et les caractères 8 à 10, les droits des autres utilisateurs (`r--`).

On peut voir que *john* a tous les droits sur son fichier, que les utilisateurs du groupe ont les droits de lecture et d'exécution sur le fichier `poisson.txt` tandis que les autres utilisateurs n'ont que le droit de lecture.

Outre les droits d'accès, l'option `-l` affiche le nom du propriétaire (john), le nom du groupe propriétaire (python), la taille du fichier en octets (8) et le jour et la date de dernière modification du fichier. Si le nom du propriétaire ou du groupe est trop long, seuls les huit premiers caractères sont affichés.

3.2 Cas d'un répertoire

Les mêmes droits existent pour les répertoires avec cependant quelques nuances dans leur signification :

— le droit de lecture signifie qu'on peut lister le contenu, c'est-à-dire obtenir la liste des fichiers présents dans le répertoire ;
— le droit d'écriture permet d'ajouter et de supprimer des fichiers dans ce répertoire et de renommer les fichiers qu'il contient ;
— le droit d'exécution signifie que l'utilisateur a le droit de se positionner dans le répertoire, avec la commande `cd` par exemple.

3.3 Changer les droits des éléments du système de fichiers

La commande `chmod` (*CHange MODe*) permet de modifier les droits d'accès d'un fichier ou d'un répertoire. Pour pouvoir l'utiliser, il faut être le propriétaire du fichier. La syntaxe générale est la suivante :

`chmod` *public opération droit*

où :
— *public* peut prendre comme valeur `u` pour le propriétaire (*user*), `g` pour le groupe, `o` pour les autres utilisateurs (*other*) ou n'importe quelle combinaison de ces trois valeurs. Il peut aussi prendre la valeur `a` (*all*) qui est équivalente à la combinaison `ugo`. Si le public n'est pas renseigné il vaut par défaut `ugo`.
— *opération* est le caractère `'+'` pour ajouter le droit, `'-'` pour le supprimer ou `'='` pour affecter un droit.
— *droit* peut prendre comme valeur `r`, `w`, `x` ou toute combinaison de ces trois valeurs.

Par exemple, la commande suivante ajoute les droits d'écriture et d'exécution pour le groupe et les autres utilisateurs :

```
$ ls -l poisson.txt
-rwxr-xr--    1 john    python    8 2017-02-12 08:23 poisson.txt
$ chmod go+wx poisson.txt
$ ls -l poisson.txt
-rwxrwxrwx    1 john    python    8 2017-02-12 08:23 poisson.txt
```

La commande qui suit supprime le droit de lecture pour les autres utilisateurs :

```
$ chmod o-r poisson.txt
$ ls -l poisson.txt
-rwxrwx-wx    1 john    python    8 2017-02-12 08:23 poisson.txt
```

Ci-dessous, on affecte les droits `r` et `x` pour le groupe et les autres utilisateurs :

```
$ chmod go=rx poisson.txt
$ ls -l poisson.txt
-rwxr-xr-x    1 john    python    8 2017-02-12 08:23 poisson.txt
```

L'exemple suivant est un peu plus compliqué, il affecte au groupe les mêmes droits que ceux du propriétaire sauf le droit d'exécution :

```
$ chmod g=u-x poisson.txt
$ ls -l poisson.txt
-rwxrw-r-x    1 john    python    8 2017-02-12 08:23 poisson.txt
```

Enfin, on peut combiner dans une seule commande des modifications consécutives en les séparant par des virgules. Par exemple, la commande suivante affecte les droits de lecture et d'exécution à tout le monde en utilisant la cible `a` mais le droit d'écriture seulement pour le propriétaire :

```
$ chmod a=rwx,og-w poisson.txt
$ ls -l poisson.txt
-rwxr-xr-x    1 john    python    8 2017-02-12 08:23 poisson.txt
```

Il existe une autre méthode pour utiliser la commande `chmod` qui utilise le système de notation octale pour représenter les droits. Dans ce système, on considère que $r = 4$, $w = 2$ et $x = 1$. De cette façon, en ajoutant les droits, on obtient une valeur comprise en 0 (aucun droit) et 7 ($7 = 4+2+1$, c'est-à-dire tous les droits) et cela pour chacun des publics : propriétaire, groupe et autres.

Avec cette notation les droits `rwxr-xr--` sont équivalents à `754` comme indiqué par la figure 1.4.4. Pour affecter ces droits sur le fichier `poisson.txt`, on effectuera la commande `chmod 754 poisson.txt`. La méthode octale ne permet pas d'enlever ou d'ajouter un type de droit. Elle sert à affecter des droits. Les anciens droits sont écrasés. Après un petit calcul mental, elle peut paraître plus rapide.

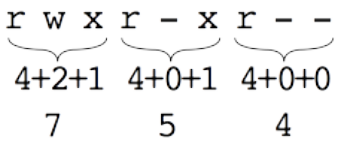


FIGURE 1.4.4 – Expression des droits en notation octale.

✍ **Challenge C14Q3**

Éléments des prochaines missions

✍ **Challenge C14Q4**

Accéder aux instructions du challenge C14Q5

3.4 Changer les droits par défaut, la commande `umask`

La commande `umask` (*User MASK)* permet de changer les droits attribués par défaut. Elle prend en argument un masque constitué de trois valeurs octales qui détermine les droits à supprimer lors de la création d'un fichier par rapport aux droits qui lui sont attribués par défaut, à savoir 666 pour les fichiers ordinaires et 777 pour les répertoires.

Sur la plupart des systèmes le masque par défaut est 022, c'est-à-dire qu'il impose lors de la création d'un fichier **de ne pas attribuer** le droit d'écriture (2) pour le groupe et pour les autres utilisateurs. Ainsi, lors de la création d'un fichier ordinaire qui par défaut devrait avoir les droits 666 soit `rw-rw-rw-`, le mask 022 empêche de donner le droit w pour le groupe et les autres utilisateur et le fichier est finalement créé avec les droits `rw-r--r--` , soit 644. De manière analogue, lors de la création d'un répertoire qui devrait avoir les droits 777 soit `rwxrwxrwx`, le mask 022 a pour effet de le créer avec les droits `rwxr-xr-x` soit 755.

Plus précisément le système calcule les droits à affecter à un nouveau fichier par une opération logique bit à bit entre le mask et les droits : étant donné un masque M et des droits par défaut D, les droits attribués au fichier créé sont le résultat de l'opération logique bit à bit `NOT(M) AND D`. Pour visualiser cette opération il est nécessaire d'écrire `M` et `D` dans leur représentation binaire, chaque bit indiquant la présence (1) ou l'absence (0) d'un droit.

Ainsi en prenant comme masque `M` = 022 et comme droits par défaut `D` = 666 pour la création d'un fichier ordinaire, on a :

— la représentation binaire de `M` : 000 010 010 (qui correspond aux droits `--- -w- -w-` à enlever)
— la représentation binaire de `D` : 110 110 110 (qui correspond aux droits `rw- rw- rw-`)
— la négation bit à bit de `M` : `NOT(M)` = 111 101 101
— et les droits attribués sont déterminés par l'opération `NOT(M) AND` D qui donne 110 100 100 (qui correspond aux droits `rw- r-- r--`)

	r	w	x	r	w	x	r	w	x
M	0	0	0	0	1	0	0	1	0
NOT(M)	1	1	1	1	0	1	1	0	1
D	1	1	0	1	1	0	1	1	0
NOT(M) AND D	1	1	0	1	0	0	1	0	0

En appliquant de manière analogue le masque 022 à des droits par défaut 777 pour la création d'un répertoire on obtient les droits 755, soit `rwxr-xr-x`.

Notez que, pour des raisons de sécurité, avec les droits par défaut 666 `umask` ne permet pas de créer des fichiers ordinaires exécutables. Après leur création, il vous faudra ajouter explicitement les droits en exécution pour chaque fichier.

Dans l'exemple suivant, on crée un fichier ordinaire vide par la commande `touch`.

```
$ touch fichier1
$ ls -l
-rw-r--r--    1 john    python     0  2017-02-12 10:09 fichier1
$ mkdir rep1
$ ls -l
-rw-r--r--    1 john    python     0  2017-02-12 10:09 fichier1
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
$ umask 027
$ touch fichier2
$ ls -l
-rw-r--r--    1 john    python     0  2017-02-12 10:09 fichier1
-rw-r-----    1 john    python     0  2017-02-12 10:10 fichier2
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
$ mkdir rep2
$ ls -l
-rw-r--r--    1 john    python     0  2017-02-12 10:09 fichier1
-rw-r-----    1 john    python     0  2017-02-12 10:10 fichier2
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
drwxr-x---    2 john    python  4096  2017-02-12 10:12 rep2/
```

On peut voir sur cet exemple que le fichier `fichier1` et le répertoire `rep1` se voient attribuer les droits par défaut déterminés par le masque 022. Après l'exécution de la commande `umask 027`, les nouveaux fichiers et répertoires ont respectivement les droits 640 et 750, ce qu'on observe pour `fichier2` et `rep2`.

Remarquons aussi que les droits des fichiers précédemment créés ne sont pas modifiés. En effet, la commande `umask` n'a pas d'effet rétroactif.

4 Les commandes de changement d'appartenance

Par défaut, le propriétaire d'un fichier est l'utilisateur qui le crée et le groupe auquel appartient ce fichier est le groupe principal du propriétaire.

La commande **chown** (*CHange OWNer*) permet de changer le propriétaire d'un fichier. Elle s'utilise de la manière suivante, en indiquant pour l'utilisateur soit l'identifiant de compte (*login*), soit l'`uid` :

`chown` *utilisateur fichier*

Sur la plupart des systèmes, seul le super-utilisateur (*root*) peut changer le propriétaire d'un fichier. Les utilisateurs ordinaires n'ont en général pas le droit de « céder » un fichier à un autre utilisateur, cela afin d'éviter les problèmes que cela pourrait occasionner dans la bonne gestion du système. Par exemple, sur un système où un quota de disque est octroyé à chaque utilisateur, la possibilité pour un utilisateur de céder la propriété d'un fichier pourrait lui permettre de détourner ces quotas.

La commande **chgrp** (*CHange GRouP*) permet de changer le groupe auquel appartient un fichier. Elle s'utilise de la manière suivante, en indiquant pour le groupe soit le nom, soit le `gid` :

`chgrp` *groupe fichier*

Seuls le super-utilisateur et le propriétaire du fichier peuvent en changer le groupe.

Appliquées sur un répertoire avec l'option `-R`, ces commandes changent le propriétaire ou le groupe du répertoire en question ainsi que, récursivement, tous les fichiers et répertoires qu'il contient.

Après exécution d'une commande **chown** ou **chgrp**, les droits du fichier ne sont pas modifiés mais ils concernent le nouveau propriétaire et/ou les utilisateurs du nouveau groupe.

```
$ ls -l
-rw-rw-r--    1 john    python     8 2017-02-12 08:23 poisson.txt
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
drwxr-x---    2 john    python  4096  2017-02-12 10:12 rep2/
$ chown wanda poisson.txt
$ ls -l
-rw-rw-r--    1 wanda   python     8 2017-02-12 08:23 poisson.txt
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
drwxr-x---    2 john    python  4096  2017-02-12 10:12 rep2/
```

Dans l'exemple ci-dessus, après exécution de la commande **chown**, le propriétaire du ficher `poisson.txt` est l'utilisateur *wanda* qui a les droits de lecture et d'écriture sur ce fichier (même s'il n'appartient pas au groupe *python*). L'utilisateur *john* a lui aussi les droits en lecture et en écriture car il est membre du groupe *python*. En revanche, il n'a plus la possibilité de changer le groupe propriétaire du fichier `poisson.txt`. Seuls *wanda* et le super-utilisateur peuvent le faire.

```
$ chgrp circus rep2
$ ls -l
-rw-rw-r--    1 wanda   python     8 2017-02-12 08:23 poisson.txt
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
drwxr-x---    2 john    circus  4096  2017-02-12 10:12 rep2/
```

Après l'exécution de la commande `chgrp circus rep2`, les membres du groupe *python* n'ont plus accès au répertoire `rep2` car il sont considérés comme faisant partie du public *autre* pour lequel aucun droit n'est accordé. Bien sûr *john*, bien que membre du groupe *python*, conserve l'accès au répertoire puisqu'il en est propriétaire.

```
$ chown wanda rep2
$ ls -l
-rw-rw-r--    1 wanda   python     8 2017-02-12 08:23 poisson.txt
drwxr-xr-x    2 john    python  4096  2017-02-12 10:10 rep1/
drwxr-x---    2 wanda   circus  4096  2017-02-12 10:12 rep2/
```

Cette fois, en ce qui concerne le répertoire `rep2`, *john* est considéré comme faisant partie du public des autres utilisateurs. Il a cédé la propriété du répertoire à *wanda* et n'est pas membre du groupe *circus*, il n'a donc plus aucun accès au répertoire ni à son contenu.

✍ **Challenge C14Q5**

Tout est dans c14q5.txt

5 Prendre temporairement l'identité du super-utilisateur

La commande `sudo` (*Switch User DO*) permet à un utilisateur de prendre temporairement l'identité du super-utilisateur (*root*) afin d'exécuter une commande nécessitant ses droits d'accès.

`sudo` *commande*

Par exemple le fichier `/etc/shadow` n'est accessible qu'au super-utilisateur, un utilisateur ordinaire ne peut donc pas le consulter sauf s'il est autorisé à utiliser la commande `sudo`. La commande `cat` est utilisée ici pour afficher le contenu d'un fichier. Nous reviendrons dans l'activité 1.5 sur la présentation de cette commande.

```
$ id
uid=1939(john) gid=1000(python) groupes=1000(python),4(adm),
24(cdrom),27(sudo),118(lpadmin))
$ cat /etc/shadow
cat: /etc/shadow: Permission non accordee
$ sudo cat /etc/shadow
Password:
[...]
root:x:17155:0:::::
john:x:17155:0:::::
wanda:x:17155:0:::::
[...]
$ id
uid=1939(john) gid=1000(python) groupes=1000(python),4(adm),
24(cdrom),27(sudo),118(lpadmin))
```

Lorsque vous utilisez `sudo`, vous êtes invité à saisir votre mot de passe. En effet, pour la gestion de la sécurité du système, toutes les commandes exécutées avec `sudo` sont conservées dans les fichiers journaux du système avec l'identité des utilisateurs. Par ailleurs, seuls les utilisateurs référencés dans le fichier `/etc/sudoers` sont autorisés à utiliser `sudo`, avec plus ou moins de restrictions sur les actions qu'ils peuvent réaliser.

Par défaut, sur une machine tournant sous un système de type Unix, le premier utilisateur créé lors de l'installation (en général le propriétaire de la machine) est automatiquement référencé dans le fichier `/etc/sudoers` avec les mêmes autorisations que le super-utilisateur.

✍ **Challenge C14Q6**

Root

Exercice 1.4.1: Partage de fichiers entre utilisateurs
Alice a un fichier `liste-achats.txt` contenant une liste d'achats qu'elle souhaite partager avec d'autres utilisateurs, dont Bob. On sait qu'Alice a utilisé la commande `umask` avec `022` en argument. Cette configuration des droits était en application lorsque le fichier `liste-achats.txt` a été créé. Le groupe par défaut d'Alice est `user`, qui est un groupe commun à tous les utilisateurs.

- Bob peut-il lire ce fichier ?
- Alice souhaite créer un groupe pour contrôler finement qui a accès au fichier. Pour ce faire, elle ajoutera une ligne au fichier `/etc/group`. Peut-elle écrire directement dans ce fichier ? Doit-elle demander à avoir les droits super-utilisateur ? .
- Donner la ligne qu'Alice doit ajouter au fichier `/etc/group` pour créer le groupe achat, y associer le gid 1002 et y ajouter *bob* et *alice*.
- Elle souhaite changer le groupe du fichier `liste-achats.txt` et l'associer au groupe achat. Donner la commande permettant de réaliser cela.
- À ce stade, quels sont les droits de Bob sur `liste-achats.txt` ? Peut-il consulter et modifier ce fichier ?
- Un autre utilisateur nommé Wanda peut-il consulter le fichier ?
- Donner une commande qui permette de définir les droits d'accès suivants sur le fichier `liste-achats.txt` : le propriétaire (Alice) a les droits de lecture et écriture (pas en exécution puisqu'il s'agit d'une liste de course) ; de même, les membres du groupe `achat` ne peuvent que lire et écrire dans ce fichier ; les autres utilisateurs n'ont aucun droit.

Solution page 61

6 Conclusion

On a vu dans cette activité que pour pouvoir utiliser un système de type Unix, un utilisateur doit posséder un compte défini par un identifiant de compte (*login*) et un mot de passe. Il existe aussi un super-utilisateur (*root*) qui administre le système et qui a tous les droits sur le système.

Les utilisateurs sont répartis dans des groupes. Les droits d'accès aux fichiers en lecture (`r`), écriture (`w`) et exécution (`x`) sont définis pour trois publics : le propriétaire, les membres du groupe et les autres utilisateurs. Le propriétaire d'un fichier peut en modifier les droits d'accès avec la commande `chmod`, et changer le groupe propriétaire avec `chgrp`. Il est aussi possible de modifier le propriétaire d'un fichier avec la commande `chown` mais celle-ci n'est en général exécutable que par le super-utilisateur.

Pour certaines commandes nécessitant d'être le super-utilisateur, et à condition qu'il soit autorisé à le faire, un utilisateur peut prendre l'identité *root* le temps de l'exécution d'une commande en la faisant précéder par la commande `sudo`.

Solutions des exercices

Solution de l'exercice 1.4.1 page 60:
Le masque par défaut étant 022, on sait que le fichier `liste-achats.txt` aura pour droit 644. En effet, la commande **`umask`** définit les droits par défaut comme étant la différence entre 666 et le masque spécifié par la commande **`umask`**. La différence (chiffre à chiffre) entre 666 et 022 vaut 644. Le droit par défaut est donc `rw-r--r--`.

- Le droit du fichier étant `rw-r--r--`, Bob y a accès parce tout le monde peut lire le fichier. De plus, il fait partie du groupe **`user`**, groupe par défaut des utilisateurs, à qui appartient le fichier et qui y a accès en lecture.
- Alice ne peut pas écrire directement dans ce fichier puisqu'il appartient au super-utilisateur root avec les droits `-rwxrwxr-x`. Elle ne fait pas partie du groupe du super-utilisateur. Pour écrire dans ce fichier, elle doit donc prendre les droits super-utilisateur en préfixant la commande d'édition par **`sudo`** ou en ouvrant une session bash avec l'identifiant root (`su root`). Notons que Alice aurait pu utiliser la commande **`groupadd`**, cependant, celle-ci n'est pas incluse dans la weblinux.
- Pour créer le groupe **`achat`** avec le **`gid`** 1002, avec comme membre *alice* et *bob*, elle doit ajouter la ligne `achat:x:1002:alice,bob`.
- Pour changer le groupe auquel appartient le fichier `liste-achats.txt`, elle utilise la commande **`chgrp achat liste-achats.txt`**.
- À ce stade, les droits du fichiers n'ont pas changé (`rw-r--r--`) et Bob n'y a toujours pas accès en écriture.
- Pour les mêmes raisons, tous les utilisateurs peuvent consulter le fichier `liste-achats.txt`.
- `chmod 660 liste-achats.txt`.

Activité 1.5

Traitement des fichiers de texte

1 Introduction

Les fichiers sont une série d'octets stockés sur un support numérique et identifiés par un chemin d'accès dans l'arborescence du système de fichier. Quand ces fichiers contiennent du texte ou des caractères alphanumériques, l'utilisateur peut afficher ou modifier ces fichiers. Dans cette activité, nous distinguerons les trois fonctionnalités suivantes et aborderons les commandes qui leur sont associées :

- afficher tout ou partie d'un fichier à l'écran : `cat`, `head`, `tail` ;
- consulter et naviguer dans un fichier sans l'éditer : `more`, `less` ;
- naviguer dans un fichier et l'éditer : `nano`, `vi`, `vim` ou `emacs`.

Notons que d'autres commandes permettent de modifier le contenu d'un fichier mais pas de le consulter comme par exemple `sed`, `cut`. Ces commandes seront abordées dans les activités 3.2 et 3.3.

2 Afficher un fichier dans le terminal

2.1 La commande `cat`

La commande `cat` (*conCATenate*) est la plus simple à utiliser. Sa mission est de lire le contenu des fichiers qui lui sont passés en arguments et de les afficher en continu à l'écran du terminal.

Soient le fichier `/home/alice/un-cinq.txt` de contenu "12345" et le fichier `/home/alice/six-neuf.txt` de contenu "6789". Le comportement de `cat` sera le suivant :

```
$ cat un-cinq.txt
12345
```

```
$ cat un-cinq.txt six-neuf.txt
123456789
```

```
$ cat six-neuf.txt
6789
```

```
$ cat six-neuf.txt un-cinq.txt six-neuf.txt
6789123456789
```

Notons que `cat` possède des options d'affichage. Par exemple, l'option `-n` préfixe chaque ligne par son numéro.

```
$ cat fichier-multiligne.txt
Guillaume
Alice
Bob
John
Pascal
Denis
```

```
$ cat -n fichier-multiligne.txt
  1    Guillaume
  2    Alice
  3    Bob
  4    John
  5    Pascal
  6    Denis
```

2.2 Affichage de début et fin d'un fichier : `head` et `tail`

Lorsque la taille d'un fichier dépasse les quelques dizaines de lignes, cela devient très compliqué de le consulter à l'aide de la commande `cat`. La commande `head` (respectivement `tail`) restreint la lecture du fichier à ses premières (respectivement dernières) lignes ou caractères.

Par défaut, la commande `head` affiche les dix premières lignes. Prenons l'exemple du fichier `/var/log/syslog`, le fichier de journalisation des événements qui se produisent sur votre machine. Ce fichier contient généralement plusieurs milliers de lignes et plusieurs entrées sont ajoutées chaque minute :

```
$ head /var/log/syslog
Apr  4 18:39:58 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:39:59 kernel[0]: IO80211ScanManager::getScanResult: All ...
Apr  4 18:39:59 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:39:59 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:40:00 kernel[0]: IO80211ScanManager::getScanResult: All ...
Apr  4 18:40:00 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:40:00 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:40:01 kernel[0]: IO80211ScanManager::getScanResult: All ...
Apr  4 18:40:02 kernel[0]: IO80211ScanManager::startScan: Broadc ...
Apr  4 18:40:02 kernel[0]: IO80211ScanManager::getScanResult: All ...
```

L'option `-n` spécifie le nombre de lignes affichées :

```
$ head -n 2 /var/log/syslog
Apr  4 18:39:58 kernel[0]: IO80211ScanManager::startScanMultiple: ...
Apr  4 18:39:59 kernel[0]: IO80211ScanManager::getScanResult: All ...
```

L'option `-c` limite le nombre d'octets (caractères) affichés :

```
$ head -c 10 /var/log/syslog
Apr  4 18:
```

De façon similaire, la commande `tail` affiche les dernières lignes via l'option `-n` :

```
$ tail -n 2 /var/log/syslog
Apr  5 00:00:06 com.apple.AddressBook.InternetAccountsBridge[639 ...
Apr  5 00:00:06 sandboxd[4519] ([63980]): com.apple.Addres(63980) ...
```

Avec l'option `-f`, on peut spécifier à la commande `tail` de continuer à lire et afficher le contenu d'un fichier, bien qu'il ait atteint la fin de ce dernier. Cela signifie que `tail -f` attend que du nouveau contenu soit ajouté au fichier pour l'afficher. La commande `tail -f` ne se termine pas d'elle-même et l'utilisateur doit l'arrêter explicitement via les touches CTRL+c (nous verrons plus tard l'utilité et le fonctionnement de CTRL+c). Ce comportement est particulièrement utile dans le cas où d'autres

applications ajoutent du contenu en permanence dans le fichier. C'est notamment le cas des fichiers de journalisation d'événements tels que `/var/log/system.log`.

Les trois commandes `cat`, `head` et `tail` servent à afficher rapidement et simplement le contenu d'un fichier. Cependant, lorsque la taille du fichier est importante, cela devient peu évident de consulter le contenu dans une console. Les commandes `less` et `more` sont alors d'une grande aide.

✍ **Challenge C15Q1**

Affichage

3 Consulter et naviguer dans un fichier avec `less` et `more`

Il y a une distinction claire entre consultation et édition d'un fichier. En effet, les commandes qui ne font que de la consultation sont plus simples d'utilisation, elles mobilisent moins de ressources et elles évitent également des erreurs de manipulations embarrassantes telles que la suppression ou la modification involontaires de données.

3.1 Consultation avec `more`

La commande `more` est utilisée pour consulter (mais pas modifier) le contenu d'un fichier texte, page par page. La « page » courante (la partie du fichier affichée à l'écran) démarre de la ligne courante et affiche le contenu du fichier jusqu'à remplir l'écran du terminal. La commande n'affiche pas plus de texte que l'écran ne peut en contenir, c'est pourquoi on utilise le terme « page ». Lorsque l'utilisateur appuie sur la flèche du bas du clavier ↓ (ou sur Espace ou Entrée) la ligne suivante devient la ligne courante et la page affichée est mise à jour. Lorsque la fin du fichier est atteinte, la commande se termine.

Initialement, la commande `more` ne pouvait effectuer le défilement du fichier que dans une direction unique (du début vers la fin). Si vous aviez fait défiler le fichier jusqu'à sa moitié, il n'était pas possible de revenir au début ou au tiers du fichier. Il fallait quitter et relancer la commande. Un utilisateur excédé par cette limitation a fini par créer une commande plus riche appelée `less`. Il a publié le code et elle est désormais disponible sur la totalité des systèmes de type Unix. De nos jours, la commande `more` est toujours disponible mais en réalité, elle fait appel à la commande `less`.

3.2 Faire plus avec `less`

Par rapport à la commande `more`, la commande `less` ajoute la possibilité de faire défiler un fichier à l'écran dans les deux directions. Elle propose d'autres possibilités de navigation auxquelles on peut accéder via des commandes internes. La commande interne la plus importante lors d'une première utilisation est sans doute celle qui permet de quitter le programme `less`, c'est-à-dire `q`.

La lecture d'un fichier se fait en appelant la commande **`less`** avec pour argument le chemin d'accès au fichier, qu'il soit relatif ou absolu :

```
$ less monfichier.txt
$ less /home/bob/monfichier.txt
```

Les options

Le `man` vous indiquera l'ensemble des options qui peuvent être passées à la commande `less`. Les plus couramment utilisées sont `-N` qui permet de préfixer chaque ligne affichée par son numéro de ligne et `-S` qui permet d'afficher chaque ligne du fichier sur exactement une ligne de l'écran. En effet, certains fichiers ont des lignes très longues qui compliquent la lecture lorsqu'elles sont affichées intégralement. Ce qui dépasse de la ligne de l'écran est tronqué.

Les commandes internes de `less`

Lorsque que vous visualisez un fichier avec `less`, vous pouvez saisir des commandes internes qui vous permettront de faire défiler le fichier ou encore de rechercher du contenu. La commande la plus importante est `h`, celle qui permet d'accéder à l'aide interne de `less`.

Le tableau suivant reprend les commandes utiles pour le déplacement dans le ficher. Elles peuvent être préfixées d'un nombre N.

`↓` ou `↑`	Défilement d'une ligne (ou N lignes) vers le haut ou vers le bas.
`f` ou `b`	Défilement d'un écran entier (ou N écrans) vers la fin ou le début.
`g`	Afficher le début (ou la Nième ligne) du fichier.
`G`	Afficher la fin du fichier.

Le tableau suivant reprend les commandes utiles pour la recherche dans le fichier. Elles peuvent être préfixées d'un nombre N.

`/motif`	Recherche et affiche la première (ou Nième) occurrence de la chaîne de caractères "motif" entre la position courante et la fin du fichier.
`?motif`	Recherche et affiche la première (ou Nième) occurrence du motif en remontant vers le début du fichier et en partant de la ligne courante.
`&motif`	Sélectionne et affiche les lignes qui contiennent le motif.

Notons également que l'ensemble des options qui peuvent être passées en arguments à la commande `less` peuvent également être appliquées à travers les commandes internes.

✍ **Challenge C15Q2**

Recherche dans un fichier

4 Éditer un fichier

Les éditeurs de texte les plus répandus sont `vi`, `vim`, `nano` et `emacs`. Ils sont probablement déjà installés sur votre machine quand elle fonctionne avec un système de type Unix. Nous allons nous concentrer sur `vi` et `vim` qui sont ceux qui présentent le meilleur compromis entre puissance et facilité d'utilisation.

`vi` est l'éditeur de texte le plus puissant accessible sur votre terminal. Il se prononce « vie ail » dont l'acronyme signifie *VIsual editor*. Bien qu'il puisse paraître difficile d'accès aux non-initiés, il est relativement simple dès que l'on sait qu'il se compose d'un mode insertion de texte et d'un mode commande. Lorsque `vi` est en mode insertion, le texte saisi s'insère dans le fichier. En mode commande, des commandes internes sont exécutées par l'appui de touches. Une fois que vous en aurez compris les principes de base, vous ne voudrez plus revenir à un éditeur plus simple (tel que `nano`). L'éditeur `vim`

est une version étendue de `vi`. Dans la plupart des systèmes d'exploitation, la commande `vi` appelle en réalité le programme `vim`. Ainsi, nous ne ferons pas de distinction entre `vi` et `vim`.

Notez toutefois que pour des raisons de rapidité d'exécution de la Weblinux, nous avons préféré y distinguer `vi` (plus simple et plus rapide) de `vim` (plus complet). Ainsi, pour une édition simple et rapide d'un fichier, vous pourrez utiliser `vi`. Si vous souhaitez avoir accès à des fonctionnalités étendues, vous utiliserez la commande `vim`, néanmoins celle-ci prendra quelques secondes pour se charger. Enfin, notez qu'afin de minimiser la taille de l'image de la Weblinux, l'aide interne de `vi`/`vim` n'y est pas disponible. Cependant, l'ensemble des informations et commandes utiles aux quiz et challenges sont dans la vidéo et dans ce document.

4.1 Éditer ou créer un fichier

Pour éditer ou créer un fichier, vous pouvez simplement utiliser la commande `vi` avec en argument le nom du fichier.

```
$ vi monfichier.txt
```

Au démarrage, `vi` fonctionne en mode commande. Aussi une fois dans l'éditeur, vous pouvez consulter l'aide interne par la commande `:h` suivi de [⏎]. À retenir, les commandes qui commencent par le caractère `':'` demandent à être validées par un retour à la ligne. Le basculement du mode insertion au mode commande se fait à l'aide de la touche [Esc].

Les commandes indispensables pour enregistrer et quitter l'éditeur sont les suivantes :

[Esc] :w [⏎]	sauver les modifications
[Esc] :q [⏎]	quitter l'éditeur
[Esc] :x [⏎]	sauver puis quitter
[Esc] :q ! [⏎]	quitter l'éditeur sans enregistrer les modifications

Les commandes de navigation dans le fichier sont similaires à celles de `less`. Certaines peuvent être préfixées par un nombre N pour être répétées. Les principales commandes sont :

[←] ou [→]	Déplacement du curseur d'un caractère à gauche ou à droite.
[↓] ou [↑]	Déplacement du curseur d'une ligne vers le haut ou vers le bas. Peut être préfixée par N.
[0]	Placer le curseur en début de ligne.
[$]	Placer le curseur en fin de ligne.
[w]	Placer le curseur sur le mot suivant. Peut être préfixée par N.
[b]	Placer le curseur sur mot précédent. Peut être préfixée par N.
[G]	Placer le curseur à la fin du fichier.
N [G]	Placer le curseur sur la Nième ligne du fichier (exemple 100G).
[CTRL]+[f]	Placer le curseur sur la page suivante (*forward*).
[CTRL]+[b]	Placer le curseur sur la page précédente (*backward*).
/motif	Rechercher et placer le curseur sur la première (ou Nième) occurrence de "motif" entre la position courante et la fin du fichier.
?motif	Rechercher et placer le curseur sur la première (ou Nième) occurrence de "motif" en direction du début de fichier.

Une autre commande fréquemment utilisée est `:set number` pour préfixer chaque ligne par son numéro.

✍ **Challenge C15Q3**

Mode commande de `vi`

4.2 Éditer le texte

Les actions d'édition d'un fichier sont variées. Elle comportent aussi bien l'insertion/modification de texte, que la suppression, le copier-coller ou la recherche et le remplacement de motifs complexes. `vi` distingue ces fonctionnalités et fournit des commandes pour chacune d'entre elles.

Mode insertion

Lorsque vous avez plusieurs caractères à insérer, vous pouvez entrer dans le mode insertion. Pour cela, vous devez au préalable être en mode commande (tapez Esc si vous n'y êtes pas encore). On accède au mode insertion par les commandes suivantes :

i	Insérer des caractères avant le curseur (*insert*).
a	Ajouter des caractères après le curseur (*append*).
c w	Changer les caractères du curseur jusqu'à la fin du mot courant (*change word*). Cette commande peut être préfixée par le nombre de mots à supprimer.
o	Ajouter une ligne après la ligne courante.
O	Insérer une ligne avant la ligne courante.

Lorsque vous êtes en mode insertion, vous pouvez vous déplacer avec les touches ← et →. N'oubliez pas que pour revenir au mode commande il vous suffit d'appuyer sur Esc. Parfois vous n'avez qu'un caractère à remplacer, pour cela vous pouvez utiliser la commande r suivie du caractère par lequel devra être remplacé celui qui se trouve à la position du curseur.

Suppression de texte

Il existe de nombreuses commandes pour supprimer du texte :

x	Supprimer le caractère à la position du curseur. Peut être préfixée par N, le nombre de caractères à supprimer.
d w	Supprimer un mot (*delete word*). Peut être préfixée par N.
d N d	Supprimer N lignes à partir de la ligne courante. Si vous ne spécifiez pas de valeur de N, une seule ligne sera supprimée.

Copier-coller

Lorsque vous supprimez du texte, celui-ci est copié dans un presse-papier. Vous pouvez coller le contenu de ce presse-papier grâce à la commande p. Le contenu du presse-papier sera inséré après le curseur.

Si vous souhaitez coller du texte pour qu'il soit dans le presse-papier, mais sans avoir à le supprimer, vous pouvez utiliser la commande y :

y w	Copier un mot. Peut être préfixée par N.
y N y	Copier N lignes à partir de la ligne courante. Si vous ne spécifiez pas de valeur de N, une seule ligne sera copiée.

Presse-papiers supplémentaires pour le copier-coller

Notez qu'en plus du presse-papier par défaut, il y a 26 presse-papiers nommés de `a` à `z`. Pour y accéder, vous devez préfixer vos commandes avec le caractère `'"'` suivi de la lettre correspondante au presse-papier souhaité. Par exemple, l'enchaînement des touches ["][f][y]3[y] permet de copier trois lignes vers le presse-papier `f`. La commande ["][f][p] permet de coller le contenu du presse-papier `f`.

Quelques commandes utiles

En plus des commandes précédemment citées, les commandes suivantes sont bien utiles :

[.]	Répéter la commande précédente.
[u]	Annuler la commande précédente ou l'édition précédente (ligne par ligne).
[J]	Fusionner une ligne et la ligne suivante sur une même ligne. Cette commande peut être préfixée par N.
`:s`	Substituer un motif à un autre.

Pour finir ce rapide tour d'horizon, il faut citer la commande de substitution :

```
:%s/ancienmotif/nouveaumotif/g
```

Elle remplace toutes les occurrences de `ancienmotif` par `nouveaumotif` dans l'ensemble du fichier. L'omission du caractère `'%'` précédant le `s` restreint le remplacement à la ligne courante.

Attention à l'ordre des motifs dans cette commande : on substitue `nouveaumotif` à `ancienmotif` mais l'ordre d'écriture vient de l'anglais *« substitute ancientmotif by nouveaumotif »*. Pour éviter les erreurs, pensez à remplacer plutôt qu'à substituer.

Exercice 1.5.1: Substitution
Alice a trouvé le fichier contenant l'attestation du MOOC-Bash de l'étudiant Riviere Evelyne. Le contenu du fichier est reproduit ci-dessous [a] :

```
Riviere Evelyne

Objet : Attestation de suivi du MOOC Bash par Riviere Evelyne

  L'équipe du MOOC Bash de l'Université de la Réunion atteste que Riviere
   Evelyne a suivi le MOOC Bash et obtenu plus de 80\% de réussite aux
  quizz, challenges et évaluations terminales.

  Riviere Evelyne est donc désormais tout à fait capable d'utiliser les
  substitutions de l'éditeur de texte VIM.

        MOOC Bash, le 26/06/19.
```

Elle souhaite y remplacer son nom et se faire passer pour une experte du Bash avant même d'avoir fini le MOOC. Elle veut cependant ne pas y passer trop de temps et décide d'utiliser les substitutions de l'éditeur `vim` pour remplacer les noms et prénoms Riviere Evelyne par les siens : Dupont Alice.

1. Proposer la substitution que peut faire Alice de manière à générer une attestation avec son nom et son prénom. Cette action doit se faire avec `vim` en mode commande et sans même passer par le mode édition.

Solution page 71

a. Ce fichier se nomme `/home/alice/Sequence1/A15/attestation.txt`.

✍ **Challenge C15Q4**

Utilisation avancée de `vi`

5 Conclusion

Dans cette activité, vous avez vu ce qu'est un fichier texte et comment gérer ces fichiers à travers des commandes dédiées. La commande `cat` affiche le contenu d'un fichier dans le terminal. Elle est très simple mais ne permet pas de naviguer dans le fichier. Vous avez appris à contourner ce défaut via la commande `less` qui affiche le fichier page par page et permet le défilement et la recherche. Enfin, pour l'édition, nous avons abordé l'éditeur `vi` qui a la particularité d'avoir deux modes qui sont insertion et commande.

Solutions des exercices

Solution de l'exercice 1.5.1 page 69:

1. Alice doit remplacer le motif « Riviere Evelyne » par « Dupont Alice ». Nous avons vu que la commande de substitution s'utilise de la manière suivante :

```
%s/ancienmotif/nouveaumotif/g
```

La commande de substitution à appliquer est donc :

```
%s/Riviere Evelyne/Dupont Alice/g
```

Conclusion

À l'issue de cette séquence, vous voilà pleinement opérationnel pour utiliser Bash afin explorer votre système et faire appel à la puissance des commandes qui s'y cachent. Vous savez maintenant :

- saisir une commande, lancer son exécution et interpréter son résultat ;
- utiliser le `man` et l'aide interne des commandes afin de connaître la syntaxe attendue pour réaliser une tâche donnée ;
- trouver les commandes qui réalisent les tâches souhaitées ;
- interpréter l'arborescence du système de fichiers, les droits qui y sont associés et agir sur ces derniers ;
- consulter et modifier les fichiers texte.

Grâce à ces notions, vous pouvez maintenant utiliser le shell Bash comme une interface utilisateur pour contrôler votre machine. Vous pouvez également parcourir le système de fichiers, modifier celui-ci et l'utiliser pour accéder à vos fichiers texte et les modifier de manière efficace. Vous avez acquis les compétences pour pouvoir travailler en ligne de commande sur vos fichiers.

Il est temps maintenant d'étendre vos compétences afin de contrôler votre environnement et interagir finement avec ce dernier. Cela va être l'objectif de la prochaine séquence.

Maintenant que vous avez brisé la glace avec votre terminal, considérez que le plus dur est fait.

Séquence 2

Interagissez avec le Bash

Introduction

Après un premier contact avec le shell Bash qui vous a permis de mieux découvrir les éléments d'un système d'exploitation, vous allez devenir plus opérationnel en parlant son langage.

Dans cette deuxième séquence, vous allez découvrir les éléments de syntaxe du shell Bash qui vont rendre l'usage de la ligne de commande plus simple.

Il ne faut pas oublier que le shell est un interpréteur d'un langage de commande. Les lignes de commande sont interprétées puis exécutées au fur et à mesure de leur lecture. L'interprétation consiste à analyser la ligne de commande afin de traiter les caractères spéciaux. Ces caractères identifient des traitements spécifiques. En mode interactif, l'utilisateur saisit et édite la ligne de commande à partir du clavier. En mode script, la ligne de commande est lue à partir d'un fichier. Nous reviendrons sur ce mode dans la dernière séquence.

Dans cette séquence, nous allons voir les différents traitements qui sont opérés par le mode interactif.

Le fonctionnement du shell suit ces étapes :

1. La saisie de ligne de commande devient possible lorsque l'invite de commande apparaît. C'est ce que vous avez vu dans la première activité de la séquence précédente.
2. La saisie est aidée par les fonctions d'édition. En effet, lorsque vous utilisez un clavier, vous serez amené à faire des fautes de frappe ou à utiliser plusieurs fois la même commande ; vous verrez dans l'activité 2.1 comment le Bash vous facilite l'interaction.
3. Dès que la touche de retour à la ligne est enfoncée, le shell analyse la ligne de commande. Cela consiste à identifier la commande et ses arguments. Les arguments peuvent être des constructions syntaxiques comme, par exemple, celle de la substitution des caractères spéciaux utilisée pour indiquer un groupe de fichiers dans un répertoire. Nous verrons dans l'activité 2.2 cette facilité d'expression.
4. Puis, l'interprétation de ces constructions est effectuée afin de les remplacer par leur résultat. Nous verrons dans l'activité 2.3 comment utiliser des variables dans la ligne de commande ou le résultat d'une commande.
5. Ensuite, l'entrée et la sortie de la commande sont définies. Dans l'activité 2.5, nous détaillerons comment alimenter en données et récupérer le résultat d'une commande.
6. Finalement, la commande avec ses arguments substitués est exécutée par l'appel du programme correspondant. Suite à l'activité 2.4, vous serez capable de contrôler l'exécution d'une commande avec la notion de processus.

Alors, prêt pour apprendre les éléments de langage du shell ?

https ://www.overleaf.com/project/5c5d7bfccc1dc076aa083b40

Activité 2.1

Aide à l'interaction

1 Introduction

Après une première semaine d'activités, vous avez découvert Bash et l'interaction avec votre système d'exploitation au travers du terminal et de la ligne de commande. Vous avez aussi certainement été confronté aux erreurs de frappe lorsque vous écrivez une commande : nom de fichier mal orthographié, mauvais nom de commande, oubli d'un argument, etc., sans compter les exemples où on vous demande de taper une commande presque identique à une précédente. Et à chaque fois, vous devez retaper la ligne de commande. C'est agaçant, c'est une perte de temps.

Heureusement, Bash fournit des aides pour la saisie et l'édition de la ligne de commande : quelques raccourcis clavier et des combinaisons de touches spéciales qui facilitent la frappe et que nous allons voir dans cette activité.

2 Édition de la ligne de commande

Raccourcis clavier et Weblinux

Certains navigateurs web utilisés pour la Weblinux peuvent intercepter pour eux-mêmes des raccourcis du Bash. C'est le cas par exemple avec Firefox qui retient le raccourci clavier CTRL+w pour fermer la fenêtre. Dans ce cas précis, la solution est d'épingler l'onglet pour rendre ce raccourci clavier inopérant pour le navigateur. Il y a des cas où cette interception rend le raccourci clavier du Bash non utilisable dans la Weblinux.

Afin de fluidifier l'écriture et l'édition de la ligne de commande, Bash fournit donc un certain nombre de raccourcis clavier à la manière de ce qu'on peut trouver dans tout éditeur de texte digne de ce nom.

Les raccourcis clavier sont introduits par les touches spéciales CTRL ou Alt. Sur certaines distributions Linux ou sur MacOS, les raccourcis habituellement accessibles via la touche Alt sont activés via la touche d'échappement Esc.

Les tableaux suivants regroupent par type de manipulation les raccourcis disponibles. Dans ces tableaux, par le terme mot il faut comprendre toutes les suites de caractères alphabétiques et numériques séparées par des espaces ou des caractères de ponctuation.

2.1 Déplacements

CTRL+a place le curseur au début de la ligne

CTRL+e place le curseur à la fin de la ligne (*End*)

CTRL+b recule d'un caractère (*Backward*)

CTRL+f avance d'un caractère (*Forward*)

Alt+b recule d'un mot
i.e. place le curseur sur la première lettre du mot sur lequel se trouve le curseur

Alt+f avance d'un mot
i.e. place le curseur après la dernière lettre du mot sur lequel se trouve le curseur

2.2 Couper / Coller

Dans les raccourcis suivants, la chaîne de caractères coupée est stockée dans un presse-papier.

CTRL+k coupe la chaîne depuis le curseur jusqu'à la fin de la ligne (*Kill*)

CTRL+u coupe la chaîne depuis le début de la ligne jusqu'au caractère qui précède le curseur

CTRL+w coupe la chaîne depuis le caractère qui précède le curseur jusqu'au début du mot (si le curseur est placé à la fin d'un mot, coupe le mot)

Alt+← identique à CTRL+w

Alt+d coupe la chaîne depuis le caractère situé sous le curseur jusqu'à la fin du mot (si le curseur est placé au début d'un mot, coupe le mot)

CTRL+y colle la chaîne du presse-papier juste avant la position du curseur

2.3 Modification

CTRL+t inverse la position des deux caractères situés avant le curseur
(pratique quand on tape par exemple, sl au lieu de ls)

Alt+t inverse la position des deux mots situés avant le curseur
(pratique lorsqu'on a inversé deux arguments d'une commande)

Alt+c met en majuscule la lettre située sous le curseur et déplace le curseur à la fin du mot
(en plaçant le curseur au début d'un mot, met la première lettre en majuscule)

Alt+l met en minuscule toutes les lettres depuis la position du curseur jusqu'à la fin du mot
(en plaçant le curseur au début d'un mot, met le mot en minuscule)

Alt+u met en majuscule toutes les lettres depuis la position du curseur jusqu'à la fin du mot
(en plaçant le curseur au début d'un mot, met le mot en majuscule)

CTRL+_ annule la dernière modification

3 Historique des commandes

Lors d'une session sur un terminal, vous serez amené à utiliser souvent les mêmes commandes ou des commandes quasi-identiques. Bash conserve par défaut l'historique des 500 dernières commandes exécutées et fournit plusieurs moyens de le consulter. L'historique est conservé dans le fichier `.bash_history` du répertoire personnel. Ainsi d'une session à une autre, vous retrouverez votre historique. L'historique est écrit dans ce fichier au moment de la fermeture de la session.

La commande `history` liste les commandes que vous avez saisies de la plus ancienne jusqu'à la plus récente. Chaque commande est précédée de son rang dans l'historique. Suivie d'un nombre, `history n` n'affiche que les n dernières commandes de l'historique, c'est-à-dire les n plus récentes. Par exemple, la commande ci-dessous affiche les trois dernières commandes dont la commande `history` .

```
$ history 3
  501  man bash
  502  clear
  503  history 3
```

3.1 Manipulation de l'historique

La commande `history` admet des options pour modifier l'historique des commandes du terminal courant. Ainsi les options suivantes font :

`history -c` efface l'historique
`history -w` *NomDeFichier* sauve l'historique courant dans *NomDeFichier*
`history -r` *NomDeFichier* ajoute les commandes contenues dans *NomDeFichier* à l'historique courant.

✍ **Challenge C21Q1**

Une autre histoire

3.2 Se déplacer dans l'historique et rappeler une commande

Les touches fléchées ↑ et ↓ permettent de naviguer dans l'historique. Au fur et à mesure du déplacement, la commande correspondant à l'endroit où on se situe dans l'historique s'affiche, il n'y a qu'à frapper ⏎ pour l'exécuter.

Les touches fléchées sont pratiques pour revenir de quelques commandes en arrière dans l'historique mais inutilisables dès lors qu'on veut rechercher une commande parmi toutes celles de l'historique.

3.3 Rechercher une commande dans l'historique

En tapant CTRL+r vous pouvez effectuer une recherche dans l'historique. Le Bash affiche alors le message :

```
(reverse-i-search)'':
```

Ce message vous invite à taper une chaîne de caractères contenue dans la commande recherchée. Il affiche alors entre les '' la commande la plus récente contenant la chaîne demandée. Des appuis successifs sur CTRL+r permettent de remonter dans l'historique des commandes qui contiennent

la chaîne demandée. Une fois que vous avez trouvé la commande voulue, vous pouvez l'éditer puis l'exécuter en tapant Entrée. Si vous voulez abandonner la recherche, faites CTRL+g.

✍ **Challenge C21Q2**

À la recherche d'une commande

4 Expansion de l'historique

Le shell offre également une construction syntaxique pour utiliser l'historique : l'expansion de l'historique. Au travers de cet élément, une commande ou une partie de commande peut être rappelée. L'expansion de l'historique est indiquée sur une ligne de commande par le caractère spécial point d'exclamation `'!'`.

4.1 Rappeler la dernière commande

Deux points d'exclamation `!!` rappellent la dernière commande exécutée. Cela peut être très utile pour rappeler la dernière commande et l'exécuter avec les droits du super-utilisateur :

```
sudo !!
```

Lorsqu'il y a une faute de frappe sur la dernière commande, la construction de remplacement rapide notée `^ancien^nouveau^` rappelle la commande en changeant le motif ancien par nouveau. Supposons que Alice veut entrer la commande `pwd` mais elle saisit `pvd`. À la ligne suivante, elle rappelle la commande en changeant le v en w.

```
$ pvd
-bash: pvd: command not found
$ ^v^w^
pwd
/home/alice/Sequence2/A21
```

4.2 Rappeler une commande en utilisant une chaîne de caractères

Un point d'exclamation suivi d'une chaîne de caractères rappellent la commande la plus récente qui commence par cette chaîne. Par exemple, pour rappeler la dernière commande utilisée pour se placer dans un répertoire, on rentrera :

```
!cd
```

Attention : il n'y a pas d'espace après le point d'exclamation.
Si vous ne voulez pas ré-exécuter une commande mais simplement la retrouver dans l'historique pour voir si c'est bien celle que vous voulez, il suffit d'ajouter `:p` (*Print*) en fin de ligne. Ainsi pour afficher la dernière commande utilisée pour se déplacer dans un répertoire, sans l'exécuter, on fera :

```
!cd:p
```

En encadrant une chaîne de caractères entre deux points d'interrogation, on peut rechercher la commande la plus récente qui contient cette chaîne. Par exemple, on rappellera la dernière commande qui contient la chaîne `to` en tapant :

```
!?to?
```

Ça pourrait être `history` ou bien `ls toto`, alors si on veut vérifier avant de l'exécuter, on l'affiche :

```
!?to?:p
```

Remarque : si vous êtes certain de vouloir rappeler et exécuter la dernière commande (sans rajouter d'autres arguments), et uniquement dans ce cas, alors le deuxième point d'interrogation n'est pas obligatoire.

4.3 Rappeler une commande par son numéro

Les commandes dans l'historique sont numérotées (la plus récente ayant le numéro le plus élevé) ce qui permet de rappeler une commande par son numéro. Par exemple, on rappellera la commande identifiée par le numéro 347 par :

```
!347
```

Cependant il faut connaître le numéro de la commande. Une utilisation certainement plus courante est de rappeler une commande en comptant à rebours. Par exemple, si vous avez exécuté trois autres commandes depuis celle que vous voulez ré-exécuter, vous taperez :

```
!-4
```

Vous indiquerez ainsi au shell de remonter de quatre lignes dans l'historique.

✍ **Challenge C21Q3**

Rappeler une commande

4.4 Réutiliser les arguments d'une commande

Le mécanisme d'historique offre aussi la possibilité de réutiliser les arguments de la dernière commande exécutée :

- `!*` correspond à tous les arguments,
- `!^` au premier argument,
- `!$` au dernier argument.

Cela peut s'avérer très utile si vous avec fait une faute de frappe dans le nom d'une commande. Imaginons par exemple que vous ayez tapé :

```
$ ehco Bash est genial
-bash: ehco: command not found
```

Vous pouvez vous rattraper en tapant :

```
$ echo !*
Bash est genial
```

Si vous ne le trouvez pas aussi génial que ça, vous pouvez aussi taper :

```
$ echo !^ est cool
Bash est cool
```

Mais tout compte fait, vous devriez plutôt taper :

```
$ echo je suis super !$ grace a !^
je suis super cool grace a Bash
```

`!^` et `!$` font respectivement référence au premier et au dernier arguments mais vous pouvez aussi rappeler plus précisément des arguments de la dernière commande avec :

`!!:n` qui fait référence au n^{e} argument. Le numéro 0 indique la commande elle-même.
`!!:n-m` qui rappelle les arguments du n^{e} au m^{e}.

```
$ echo !$ est !!:3-6 moi
Bash est super cool grace a moi
```

La réutilisation des arguments ne s'applique pas qu'à la dernière commande. Les arguments de plusieurs commandes peuvent se combiner dans une nouvelle commande. Pour cela, il faut indiquer le numéro de la commande dans l'historique et le rang du ou des arguments à réutiliser en utilisant le caractère `':'` ce qui donne l'écriture `!n:m` qui fait référence à la commande de numéro n et à l'argument à la position m.
Imaginons que l'historique de la session courante de Alice est le suivant :

```
1 cp fichier1.txt Sequence2
2 ls Sequence3
3 cat fichier1.txt
4 cd
```

En tapant

```
$ touch !2:$/!1:1
touch Sequence3/fichier1.txt
```

le fichier appelé fichier1.txt est créé dans le répertoire Sequence3.

✍ **Challenge C21Q4**

Réutiliser les arguments

4.5 Modifier les arguments

Parfois il vous arrivera de vouloir répéter une commande ou de réutiliser les arguments mais en modifiant certaines parties. Supposons par exemple que vous exécutez la commande suivante (`wc` affiche le nombre de lignes, de mots et de caractères d'un fichier) :

```
$ wc fichier1.txt
       9       443      3024 fichier1.txt
```

Vous voulez ensuite appliquer cette commande sur `fichier2.txt`. Et bien vous pouvez le faire de la manière suivante :

```
$ !!:s/1/2/
wc fichier2.txt
       7     322    2155 fichier2.txt
```

Cela mérite quelques explications : les deux points d'exclamation `!!` qui rappellent la dernière commande sont suivis de `:s` (*substitute*) qui signale que dans la commande rappelée, on veut effectuer une substitution de chaîne. On parle d'opérateur. L'opérateur `s` indique la chaîne que l'on veut remplacer et la chaîne par laquelle on veut la remplacer. L'ancienne et la nouvelle chaînes de caractères sont indiquées entre des barres obliques (`/`). Ici, on veut remplacer la chaîne `1` par la chaîne `2` donc on écrit `/1/2/`. Le shell affiche alors la commande obtenue après la substitution demandée puis l'exécute.

Dans cet exemple, la substitution `:s` s'applique sur `!!`, c'est-à-dire sur la dernière commande de l'historique. On peut bien sûr appliquer une substitution sur une commande rappelée de l'historique par son numéro. Par exemple, en tapant `!347:s/toto/titi/` on indiquerait au shell qu'on souhaite ré-exécuter la commande de numéro 347 en remplaçant la chaîne `toto` par la chaîne `titi`.

Précisons que cette substitution ne porte que sur la première occurrence de la chaîne à remplacer. Si on veut remplacer toutes les occurrences de `toto` par `titi`, il faut écrire `!347:gs/toto/titi/`, le symbole `g` signifiant que la substitution doit être générale.

Enfin, il est aussi possible d'effectuer une substitution en utilisant les arguments de la dernière commande. Par exemple, si après avoir exécuté la commande `wc` sur `fichier2.txt`, vous voulez en faire une copie dans `fichier3.txt`, vous pouvez saisir :

```
$ cp !^ !^:s/2/3/
cp fichier2.txt  fichier3.txt
```

La commande ou une partie de commande peut être modifiée lorsqu'elle est réutilisée. Comme nous venons de le voir, une substitution de chaîne de caractères peut être appliquée avec l'opérateur `s`. L'indication de l'opérateur se fait après un caractère deux points (`':'`) supplémentaire à la fin de l'expansion d'historique. Ainsi, la ligne de commande peut être composée par les arguments de plusieurs commandes contenus dans l'historique.

```
$ cp !1:1 !2:^/!1:1:s/1/3/
cp fichier1.txt  Sequence3/fichier3.txt
```

Enfin, pour terminer sur les modifications d'arguments, l'opérateur `'r'` est bien pratique lorsque l'on traite des archives qui créent un répertoire lorsque l'on procède à l'extraction des fichiers. Cet opérateur supprime le suffixe du fichier (comme `.txt`) et conserve toute la partie précédant le suffixe. Imaginons que l'archive projet.tgz contienne le répertoire projet. Après l'extraction du répertoire de l'archive et sa création dans le répertoire courant, le changement de répertoire peut se faire de la manière suivante

```
$ tar xzf projet.tgz
$ cd !!:$:r
cd projet
```

4.6 En résumé

Nous avons vu que pour rappeler une commande par le numéro de la commande dans l'historique, nous disposons de trois formes d'écriture :

`!!` rappelle la dernière commande,
`!n` rappelle la commande avec le numéro n dans l'historique,
`!-n` rappelle la commande avec n numéros de commande avant le commande courante dans l'historique.

Mais une commande peut aussi être identifiée par une recherche par correspondance avec une chaîne de caractères.

`!`*string* rappelle la dernière commande qui commence par la chaîne *string*.

`!?`*string* `?` rappelle la dernière commande qui contient la chaîne *string*.

Outre les commandes, ce sont aussi les arguments des commandes qui peuvent être rappelés dans une nouvelle commande. La désignation des arguments s'effectue en ajoutant le caractère deux points (`':'`) et un identificateur de mots après la désignation de la commande concernée. Aussi nous avons vu :

`!!:^` (ou la forme abrégée `!^`) pour désigner le premier argument de la dernière commande,
`!!:*` (ou la forme abrégée `!*`) pour désigner tous les arguments de la dernière commande,
`!!:$` (ou la forme abrégée `!$`) pour désigner le dernier argument de la dernière commande,
`!n:m` pour désigner l'argument m de la commande de numéro n.

Exercice 2.1.1: Alice gère son historique

L'historique d'Alice associé à sa session est le suivant :

```
1 ls
2 ls -l
3 cd Sequence2/A21
4 cd
5 echo archive.tar.gz
6 echo toto mimile
7 touch Sequence2/A21/fichier.txt
```

En utilisant uniquement l'expansion d'historique, indiquer les expressions que devra saisir Alice pour effectuer les actions suivantes :

1. pour créer le `fichier.txt` (avec la commande `touch`) dans le répertoire `Sequence3/A31`,
2. pour afficher le contenu du répertoire courant avec l'option `-la`,
3. pour créer le fichier `Sequence2/A21/fichier.data`,
4. pour afficher `archive`,
5. pour afficher tous les arguments de la commande numéro 6 (mais sans exécuter la commande),
6. enfin pour terminer, mettre le répertoire `Sequence2` comme répertoire courant en utilisant la commande de numéro 3. Vous pouvez consulter le `man bash` (section DEVELOPPEMENT DE L'HISTORIQUE) afin de trouver l'opérateur qui supprime tout après le dernier caractère `/` (ou garde tout jusqu'à rencontrer le dernier caractère `/`).

Solution page 88

5 Auto-complétion

En plus de l'édition de la ligne de commande et de l'historique, Bash offre une troisième forme d'aide à la saisie qui est la plus utilisée : l'auto-complétion. Celle-ci s'effectue au moyen de la touche de tabulation ⇥. Après avoir tapé les premiers caractères, la touche tabulation demande à compléter automatiquement le nom de commande. S'il n'y a qu'une seule complétion possible, le Bash complète le nom de la commande en entier. Par exemple, `hist`⇥ est remplacé par `history` sur la ligne de commande.

Lorsqu'il y a une ambiguïté, Bash ne complète pas la ligne de commande après avoir tapé ⇥, il suffit alors de taper une deuxième fois la touche ⇥ pour avoir une liste des complétions possibles.

Par exemple, `ch`⇥ n'affiche rien car il existe plusieurs commandes commençant par les caractères `ch`. Un deuxième appui sur ⇥ affiche la liste de toutes les possibilités suivie de l'invite `$` et du début de

la ligne de commande que vous avez tapée.

`$ ch` ⇥ ⇥

```
chattr  chgrp   chmod   chown   chpst
chrt    chvt
$ ch
```

Si après un deuxième appui sur ⇥ rien n'apparaît, c'est qu'aucune complétion n'est possible. C'est une indication assez forte qu'il y a une erreur de saisie.

L'auto-complétion fonctionne de la même manière pour les noms de fichiers, ainsi la frappe :
`$ cd /h` ⇥ `a` ⇥ `D` ⇥ devrait vous permettre de vous placer dans le répertoire `/home/alice/Documents` en n'ayant frappé que 10 touches du clavier !

6 Encore deux raccourcis

Terminons cette (courte) activité en donnant deux derniers raccourcis clavier qui, bien que ne constituant pas une aide à la saisie de commande, peuvent s'avérer très utiles :

CTRL + l Permet d'effacer le contenu du terminal.

CTRL + c En cours de frappe, permet d'arrêter la saisie de la ligne de commande et de revenir à l'invite avec une ligne vierge.

7 Conclusion

Cette activité vous a permis de prendre connaissance de l'aide à la saisie fournie par le Bash. Plusieurs raccourcis clavier permettent de se déplacer et d'éditer la ligne de commande. L'historique intégré est pratique pour éviter de re-saisir des commandes précédemment utilisées. Enfin, vous pouvez user et abuser de l'auto-complétion pour écrire rapidement de longues lignes de commande.

Solutions des exercices

Solution de l'exercice 2.1.1 page 86:

1. Il faut changer tous les 2 en 3 de la dernière commande. `!!:gs/2/3/`
2. Rappeler la dernière commande `ls` et en ajoutant l'option `-a`. Il y a deux façons de le faire, soit par recherche `!ls -a` ou soit par le numéro d'ordre `!2 -a`.
3. Sur la commande de numéro 7, il faut supprimer `.txt` et compléter avec `.data` soit `!7:r.data`. L'opérateur `'r'` supprime `.txt` de la commande de rang 7. Il faut alors compléter la commande rappelée avec `.data`.
4. Pour l'argument, il faut supprimer les deux suffixes de l'argument de la commande numéro 5 et reprendre la commande numéro 1, soit `!1 !5:$:r:r`.
5. Cela consiste à désigner tous les arguments mais sans demander à exécuter la commande de numéro 6 soit `!6:*:p`.
6. L'opérateur `h` (non présenté dans ce document) est à utiliser dans ce cas. Ce qui donne `!3:h`.

Activité 2.2

Abréviations pour le nom des fichiers

1 Introduction

Il arrive que vous souhaitiez déplacer plusieurs fichiers de noms similaires vers le même répertoire de destination. Pour l'instant, vous n'avez d'autre choix que de répéter la commande pour chaque fichier. Dans cette activité, nous allons voir comment désigner plusieurs fichiers en une seule expression et donc une seule commande.

Pour cela, le Bash propose la substitution de nom de fichier (*Pathname Expansion*) qui utilise des caractères spéciaux. Une expression contenant ces fameux caractères sera remplacée par les noms de fichiers correspondants. Cette substitution est opérée par le Bash avant l'exécution de la commande proprement dite. Ainsi une phase d'analyse syntaxique est effectuée avant exécution pour interpréter les caractères spéciaux. Quand l'argument d'une commande est de type nom de fichier, celui-ci peut être le nom de fichier littéralement ou une expression de la substitution de nom de fichier. Dans ce dernier cas, on parlera de motif (*pattern*) pour qualifier ce type d'expression.

2 Caractères spéciaux et substitution de nom de fichier

2.1 Caractères désignant des répertoires

Vous avez déjà rencontré, lors de la dernière séquence (activité système de fichiers et répertoires), trois caractères spéciaux qui désignent des chemins ; il s'agit de `'.'`, `'..'` et `'~'`. Utilisés dans la désignation d'un nom de fichier, ces caractères seront remplacés par leur valeur avant l'exécution de la commande portant sur le fichier. On peut bien sûr les combiner et les utiliser plusieurs fois dans un nom de fichier. Voyons quelques exemples. Plaçons le répertoire courant dans le répertoire personnel de Alice appelé `alice` :

```
$ cd
```

Depuis ce répertoire, on peut lister le contenu du répertoire parent en utilisant le caractère spécial `'..'` [1] :

```
$ ls ..
alice  bob    user
```

On peut aussi lister le contenu du répertoire de l'utilisateur Bob.

1. On parle de caractère spécial pour ce qui est en fait ici composé par deux caractères points.

```
$ ls ../bob
Maildir    config     mikmodrc   vide
a.txt      gitconfig  scummvmrc
```

Supposons maintenant qu'Alice ignore quel est le répertoire courant et qu'elle veut lister le contenu du répertoire personnel de Bob, la commande suivante lui donnera satisfaction :

```
$ ls ~/../bob
Maildir    config     mikmodrc   vide
a.txt      gitconfig  scummvmrc
```

Alice indique, à partir de son répertoire personnel, de remonter d'un répertoire pour ensuite aller dans le répertoire `bob`.

Dans la commande précédente, le tilde (`'~'`) désigne le répertoire personnel de l'utilisateur, c'est-à-dire `/home/alice`. Avant d'exécuter la commande `ls`, le nom de fichier `~/../bob` sera donc remplacé par `/home/alice/../bob` puis par `/home/bob`.

Cependant le tilde peut avoir une autre valeur. En effet, s'il n'est pas immédiatement suivi d'un caractère '/', il ne désigne plus le répertoire personnel de l'utilisateur mais le répertoire parent du répertoire personnel. Ainsi Alice pourrait aussi bien utiliser la commande suivante avec le même résultat :

```
$ ls ~bob
Maildir    config     mikmodrc   vide
a.txt      gitconfig  scummvmrc
```

Cette fois, tilde désigne le répertoire parent du répertoire personnel d'Alice, c'est-à-dire `/home/` et le nom de chemin `~bob` est remplacé par `/home/bob`.

2.2 Caractères spéciaux

Pour désigner un ensemble de fichiers par un nom générique appelé formellement un motif, le shell propose des caractères spéciaux. Avec ces caractères, nous aurons la possibilité d'indiquer n'importe quel caractère ou n'importe quelle séquence de caractères ou un ensemble de caractères. Il s'agit respectivement du point d'interrogation `?`, de l'astérisque `*` et des crochets `[]`. Ici, il faut préciser ce qu'on entend par nom de fichier : tous les types d'éléments que l'on peut trouver dans une arborescence de fichiers. Nous allons présenter maintenant les règles pour écrire des motifs utilisés par le shell. Ces règles sont présentées sous la rubrique de *Pattern Matching* dans le manuel en ligne du Bash.

Tout nom de fichier qui contient un de ces caractères spéciaux est remplacé sur la ligne de commande par la liste de tous les fichiers du répertoire courant qui correspondent au motif ainsi donné. On appelle ce mécanisme la substitution du « nom de fichier » (*globbing*). Prenons l'exemple du répertoire personnel `alice`, la commande `ls f*` effectuée dans ce répertoire donne le résultat ci-dessous :

```
$ ls f*
fichier1.txt  fichier2.txt
fac:
classe.txt
```

Dans cet exemple, la commande entrée donne la liste de tous les noms de fichiers dont le nom commence par la lettre `'f'`. Ou plus exactement, elle donne le résultat de la commande `ls` appliquée à tous les fichiers dont le nom commence par la lettre `'f'`. En effet, le shell procède d'abord à la substitution des

caractères spéciaux puis exécute la ligne de commande ainsi transformée. Lorsqu'il y a un répertoire dans la liste retournée par la substitution opérée par le shell, comme c'est le cas avec le répertoire `fac`, le résultat obtenu est le contenu de ce répertoire sans rentrer dans les sous-répertoires (dans l'hypothèse que le répertoire `fac` contienne des sous-répertoires). Ainsi la commande `ls /*` ne listera que le contenu des répertoires de la racine et non pas de toute l'arborescence des fichiers. Pour bien comprendre que c'est le shell qui effectue la substitution et non la commande `ls`, recommençons la ligne de commande précédente mais en changeant la commande :

```
$ echo f*
fac fichier1.txt fichier2.txt
```

On peut voir que cela affiche tous les noms des éléments du répertoire courant commençant par la lettre `f`. Cette fois-ci, le contenu du répertoire `fac` dans le répertoire courant n'est plus affiché. Cette ligne de commande montre que c'est bien le shell qui a effectué la substitution du caractère spécial en nom de fichiers et non la commande `ls`.

Le caractère `'*'` remplace n'importe quelle séquence de caractères et cette séquence peut éventuellement être vide, comme on peut le voir dans l'exemple suivant :

```
$ echo *f*
fac fichier1.txt  fichier2.txt
```

On voit que le premier astérisque est remplacé par une séquence vide et que la liste des fichiers correspondant au motif `*f*` est la même que celle correspondant au motif `f*`. Ce qui montre que le premier astérisque a été remplacé par rien, le vide.

✍ **Challenge C22Q1**

De l'or caché.

Le point d'interrogation, quant à lui, remplace exactement un caractère :

```
$ echo fichier?.txt
fichier1.txt  fichier2.txt
```

Ici, s'il existait un `fichier12.txt`, il ne serait pas listé. Et pour afficher les noms de tous les fichiers comportant exactement cinq caractères, Alice exécutera :

```
$ echo ?????
Books Music
```

Et que se passe-t-il si aucun fichier ne correspond au motif donné ? Essayons avec la commande `ls` :

```
$ ls *X
ls: *X: No such file or directory
```

On obtient pour cet exemple une erreur. En effet, aucun fichier dans le répertoire `alice` ne correspond au motif `*X` (aucun fichier n'a un nom se terminant par la lettre X), le shell n'a donc pas pu réaliser de substitution et a laissé le nom tel quel. La commande `ls` provoque une erreur et indique que le fichier `*X` n'existe pas.

Alice essaye maintenant d'afficher les noms de tous les fichiers présents dans son répertoire personnel :

```
$ echo *
Books Compagnon Documents Movies Music Pictures Sequence1 Sequence2 Sequence3
   Sequence4
fichier1.txt fichier2.txt imdb
```

Elle est un peu déçue par le résultat car elle sait bien que son répertoire contient aussi certains fichiers « cachés » dont le nom commence par un point. En effet, si un nom de fichier est préfixé par un caractère '`.`', il ne pourra pas se substituer aux caractères spéciaux '`*`' ou '`?`' et ce, pour cause de conflit avec la signification particulière du '`.`' (qui, rappelons-le, dans l'interprétation des noms de fichier désigne le répertoire courant). Pour obtenir le résultat désiré, Alice doit donc explicitement faire figurer ce caractère dans le motif et de ce fait, doit donner deux motifs :

```
$ echo .* *
. .. .bash_history .bashrc .profile Books Compagnon Documents Movies Music Pictures
Sequence1 Sequence2 Sequence3 Sequence4 fichier1.txt fichier2.txt imdb
```

✍ **Challenge C22Q2**

Chercher des aiguilles dans une botte de foin, mais avec le bon aimant.

On peut aussi indiquer dans un motif qu'un nom de fichier doit contenir un caractère particulier parmi un ensemble de caractères. Il suffit de préciser entre crochets (`[]`) l'ensemble des caractères voulus. Attention, dans le motif, il n'y pas de caractères espace qui entourent les crochets.
Par exemple, la commande `echo [aef]*` affiche tous les fichiers qui commencent par `a` ou `e` ou `f`. Il est aussi possible de donner un intervalle de caractères.
Ainsi `echo [a-m]*` affiche tous les fichiers dont la première lettre est comprise entre `a` et `m`.
Et `echo [A-Z]*[0-9]` affiche tous les fichiers dont le nom commence par une majuscule et dont le dernier caractère est un chiffre.
Et si on fait précéder les caractères entre les crochets d'un point d'exclamation, alors cela signifie qu'on ne veut aucun des caractères entre les crochets. Par exemple, la commande `echo [!A-Z]*` affiche tous les fichiers qui ne commencent pas par une lettre majuscule.

Classes de caractères

Il est assez courant d'avoir besoin de désigner un caractère parmi un ensemble de caractères, comme les majuscules, les minuscules, les chiffres, etc ... La norme POSIX (*Portable Operating System Interface UniX*) définit un certain nombre de classes de caractères qui font référence à des sous-ensembles de caractères particuliers. Une classe est représentée par un nom qui décrit l'ensemble de caractères encadré entre '`[:`' et '`:]`'. Voici ces classes et leur signification (certaines ne sont pas vraiment destinées à un usage dans le contexte de la substitution des noms de fichiers) :

`[:alnum:]`	Correspond aux caractères alphabétiques et numériques. Équivalent à `A-Za-z0-9`.
`[:alpha:]`	Correspond aux lettres. Équivalent à `A-Za-z`.
`[:blank:]`	Correspond à une espace ou à une tabulation.
`[:cntrl:]`	Correspond aux caractères de contrôle.
`[:digit:]`	Correspond aux chiffres. Équivalent à `0-9`.
`[:graph:]`	Correspond aux caractères de code ASCII compris entre 33 et 126 . Ce sont les caractères graphiques affichables hormis l'espace.
`[:lower:]`	Correspond aux lettres minuscules. Équivalent à `a-z`.

`[:print:]`	Correspond aux caractères de code ASCII compris entre 32 et 126. C'est la classe `[:graph:]` avec l'espace en plus.
`[:space:]`	Correspond aux blancs (espace, tabulation, passage à la ligne, retour chariot, saut de page, tabulation verticale).
`[:upper:]`	Correspond aux lettres majuscules. Équivalent à `A-Z`.
`[:xdigit:]`	Correspond aux chiffres hexadécimaux. Équivalent à `0-9A-Fa-f`.

Important : Une classe correspond à un intervalle de caractères ; pour être utilisée, il faut donc la placer entre crochets ; on écrira par exemple :

```
echo [[:upper:]]*[[:digit:]]
```

ce qui est équivalent à :

```
echo [A-Z]*[0-9]
```

Les caractères spéciaux sont puissants et font gagner énormément de temps à l'utilisateur mais il faut prendre des précautions lors de leur emploi. Prenons l'exemple d'un utilisateur qui souhaite supprimer l'ensemble des fichiers et répertoires qui commencent par un `e`. Si celui-ci fait une faute de frappe, et tape `rm -rf *` au lieu de `rm -rf e*` le malheureux aura supprimé tout le contenu de son répertoire courant (sous répertoires compris) !

On pourrait imaginer que la commande `rm` demande une confirmation lorsque l'argument donné est '`*`' mais cela n'est pas possible du fait de la manière dont la ligne de commande est évaluée. En effet, comme il est dit dans l'introduction de cette activité : la substitution des caractères spéciaux est opérée par le Bash **avant** l'exécution de la commande proprement dite. Donc ce n'est pas le caractère '`*`' qui est passé comme argument à la commande `rm -rf` mais le résultat de la substitution de ce caractère, c'est-à-dire la liste du contenu du répertoire.
Moralité de l'histoire, l'usage de la commande de suppression `rm` s'utilise avec précaution surtout lorsqu'on ne demande pas de confirmation (`-f`) et récursivement (`-r`). Sinon, quand vous aurez perdu tous vos fichiers, vous pourrez vous remémorer cette citation « l'erreur est humaine mais un véritable désastre nécessite un ordinateur ».

✍ **Challenge C22Q3**

Votre fréquence cardiaque augmente avec la difficulté.

En résumé

Les caractères spéciaux de *pattern matching* utilisés pour désigner des fichiers sont résumés par le tableau 2.

	Signification
`*`	une chaîne de caractères vide ou quelconque
`?`	un caractère quelconque
`[abf]`	un caractère pris dans un ensemble
`[a-f]`	un caractère pris dans l'intervalle
`[[:classe:]]`	un caractère pris dans une classe
`[!...]`	aucun caractère de l'ensemble ou de l'intervalle ou de la classe

Tableau 2 – Caractères spéciaux pour le nom de fichiers.

Exercice 2.2.1: Alice a un caractère spécial
Dans le répertoire `Compagnon/A22` de Alice, il y a les fichiers suivants :

```
fic1.txt       fier           filer          indestructible male
fic2.txt       figuier        fin            lipsum.txt     miles
fichier        file           ile            lorem.txt      pale
```

Donner la signification de chacun des motifs ci-dessous et le résultat obtenu. Procéder à la vérification à l'aide de la commande `echo`.

1. `?ale`
2. `[f-p]?le`
3. `?ile?`
4. `fic*`
5. `fi*er`
6. `*.txt`
7. `fic[12].txt`
8. `[f-p]*le`

Solution page 97

3 Encore plus de possibilités avec l'option `extglob`

Le Bash offre des motifs encore plus élaborés avec l'option *EXTended GLOBbing*. Cette option de comportement du shell est contrôlée avec la commande `shopt`. En l'occurrence, pour bénéficier de la substitution étendue, il faut entrer la commande : `shopt -s extglob` (avec `'-s'` pour *set* et `'-u'` pour *unset*).

Cette option offre des motifs étendus composés éventuellement d'une liste de sous-motifs, c'est-à-dire de motifs séparés par des caractères `'|'`. Ce caractère donne la signification d'un 'ou' logique. Par exemple, la liste `ab|cd` indique le motif `ab` ou `cd`. Les motifs étendus sont formés avec un ou plusieurs des éléments suivants :

`?(`*liste-de-motifs*`)`
: signifie 0 ou 1 fois un des sous-motifs de la liste

`*(`*liste-de-motifs*`)`
: signifie un nombre quelconque de fois - éventuellement 0 fois - un des sous-motifs de la liste

`+(`*liste-de-motifs*`)`
: signifie au moins une fois un des sous-motifs de la liste

`@(`*liste-de-motifs*`)`
: signifie exactement un des sous-motifs de la liste

`!(`*liste-de-motifs*`)`
: signifie aucun des sous-motifs de la liste

Illustrons par quelques exemples afin de découvrir les possibilités de la substitution étendue du nom de fichier. Supposons que le répertoire courant contienne les fichiers suivants :

```
Xa      Xaaca    Xbb        Xyzaaaaba   aaXxy
Xaaa    Xaxxxb   Xbooooaa   Xyzax
```

Pour afficher les noms des fichiers dont le nom commence par `Xaa` ou `Xb` et se termine par `aa` ou `b` on écrira :

```
$ echo X@(aa|b)*@(aa|b)
Xbb Xbooooaa
```

Pour obtenir presque le même résultat sans substitution étendue, il faudrait utiliser les quatre motifs `Xaa*aa Xaa*b Xb*aa Xb*b`.

Le résultat est presque le même car si un des motifs ne correspond à aucun fichier, il n'est pas remplacé et apparaît tel quel dans le résultat comme le montre l'exemple suivant :

```
$ ls
Xa      Xaaca    Xbb        Xyzaaaaba   aaXxy
Xaaa    Xaxxxb   Xbooooaa   Xyzax
$ echo X@(aa|b)*@(aa|b)
Xbb Xbooooaa
$ echo Xaa*aa Xaa*b Xb*aa Xb*b
Xaa*aa Xaa*b Xbooooaa Xbb
```

Aucun nom de fichier ne commence par `Xaa` et se termine par `aa`, de même aucun fichier n'a un nom qui commence par `Xaa` et se termine par `b` ; les motif deux motifs `Xaa*aa` et `Xaa*b` ne correspondent à aucun fichier et sont affichés tel quel dans le résultat.

Une possibilité encore plus intéressante avec la substitution étendue est certainement la négation. Par exemple, pour obtenir la liste des fichiers dont le nom ne se termine pas par les deux caractères `aa` on écrira :

```
$ echo !(*aa)
Xa Xaaca Xaxxxb Xbb Xyzaaaaba Xyzax aaXxy
```

Attention à ne pas faire l'erreur d'écrire `echo *!(aa)` qui est équivalent à écrire `echo *`. Ce filtre signifie « n'importe quelle suite de caractères non suivie de `aa` », ce que vérifie par exemple le nom `Xaaa` ; en effet le caractère `*` est remplacé par le nom en entier qui n'est pas suivi de `aa` puisqu'il n'y a plus d'autre caractère !

Si on veut la liste des fichiers qui commencent par `X` mais pas le fichier `Xaa`, on écrira :

```
$ echo X!(aa)
Xa Xaaca Xaxxxb Xbb Xbooooaa Xyzaaaaba Xyzax
```

Et si on veut la liste des fichiers commençant par `X` et ne comprenant pas une suite de lettres `a` dans le début du nom, on écrira :

```
$ echo X!(+(a)*)
Xbb Xbooooaa Xyzaaaaba Xyzax
```

Pour terminer par un dernier exemple, si on veut n'afficher que les noms des fichiers qui contiennent exactement un caractère `a`, on écrira :

```
$ echo *([!a])@(a)*([!a])
Xa Xaxxxb Xyzax
```

Traduit en langage courant, ce motif signifie : un nom formé d'un nombre quelconque de caractères autres que le caractère `a` suivi d'exactement un caractère `a` suivi d'un nombre quelconque de caractères autres que le caractère `a`.

✍ **Challenge C22Q4**

On s'étend et on reste détendu.

4 Conclusion

Avec cette activité, vous avez pu apprendre à effectuer des opérations sur des ensembles de fichiers en une seule ligne de commande grâce au mécanisme d'interprétation des noms de fichiers intégré au shell. Ce mécanisme s'opère à travers la substitution de motifs pour obtenir des noms de fichiers. Les motifs contiennent des caractères spéciaux '`*`', '`?`' et '`[]`'. Par défaut, seuls ces trois caractères sont utilisés pour l'interprétation des noms de fichier. Ils désignent respectivement : « une suite de caractères quelconque éventuellement vide », « exactement un caractère quelconque » et « un des caractères donnés entre crochets ».

Ce comportement par défaut facilite déjà grandement la manipulation de groupes de fichiers. La substitution étendue du nom de fichier, accessible en positionnant l'option `extglob` de Bash, accroît les possibilités de traitement par lot des fichiers à l'aide de motifs plus sophistiqués mais peut-être plus délicats à mettre en œuvre et moins lisibles dix minutes après les avoir écrits···

Solutions des exercices

Solution de l'exercice 2.2.1 page 94:
La signification de chaque motif est dans l'ordre :

1. Les noms de 4 caractères qui se terminent par la chaîne 'ale'.

```
echo ?ale
male pale
```

2. Les noms commençant par une lettre comprise entre f et p de 4 lettres avec un caractère quelconque en seconde position.

```
echo [f-p]?le
file male pale
```

3. Les noms de 5 lettres avec la chaîne 'ile' au milieu.

```
echo ?ile?
filer miles
```

4. Les noms commençant par la chaîne 'fic'.

```
echo fic*
fic1.txt fic2.txt fichier
```

5. Les noms qui commencent par 'fi' et se terminent par 'er'.

```
echo fi*er
fichier fier figuier filer
```

6. Les noms qui se terminent avec le suffixe '.txt'.

```
echo *.txt
fic1.txt fic2.txt lipsum.txt lorem.txt
```

7. Les noms de fichier 'fic1.txt' et 'fic2.txt'.

```
echo fic[12].txt
fic1.txt fic2.txt
```

8. Les noms commençant par une lettre comprise entre f et p et se terminant par la chaîne 'le'.

```
echo [f-p]*le
file ile indestructible male pale
```

Activité 2.3

Constructions syntaxiques

1 Introduction

Dans cette activité, nous allons étudier les constructions syntaxiques du shell. Ces constructions sont communes à beaucoup de langages de commande. Ces constructions sont interprétées avant que la commande s'exécute. Juste après que la ligne de commande a été validée par la touche de retour à la ligne, le shell effectue l'analyse syntaxique qui a pour but d'arriver à identifier la commande et ses arguments. Il commence ainsi par identifier les mots de la ligne de commande. Un mot est une suite de caractères qui se termine par une espace, une tabulation ou un saut de ligne. Lors de cette analyse, le shell applique les substitutions qu'il reconnaît. On appelle substitution le remplacement d'un mot par l'interprétation de ce mot. Le shell offre plusieurs types de substitutions (ou `expansion`). Cependant, les substitutions restent sous le contrôle de l'utilisateur. Celui-ci peut les inhiber au moyen de caractères spéciaux. Cette activité traite d'abord de la substitution de variable et de la substitution de commande. Et se termine par les moyens offerts par le shell pour inhiber les substitutions.

2 Substitution de variable

En informatique, une variable est un espace de stockage en mémoire identifié par un nom. Cet espace de stockage sert à enregistrer une valeur. Comme tous les langages de programmation, le Bash reprend cette notion de variable avec la particularité que la variable en Bash n'est pas typée par défaut. Dans ces conditions, la variable ne peut prendre qu'un seul type de valeur : la chaîne de caractères. La conséquence de l'absence de typage est qu'une variable n'a pas besoin d'être déclarée avant son utilisation. Enfin, il faut noter que la valeur d'une variable peut être modifiée. L'affectation d'une valeur à une variable utilise le symbole `'='`. Un point important dans cette opération : il ne faut pas laisser d'espace autour du caractère égal. L'affectation s'écrit de la manière suivante :

$$Nom\_de\_la\_variable{=}valeur$$

Le nom de variable peut être formé uniquement avec des caractères alphanumériques (A-Z, a-z et 0-9) et le caractère souligné (`'_'`). Par exemple, nous pouvons définir une variable qui contient le caractère '1' et une autre variable qui contient un chemin d'accès.

```
i=1
mon_cv=Documents/cv
```

Pour retrouver la valeur d'une variable, il faut préfixer le nom de la variable par `$`. Par exemple, pour l'affichage du contenu de la variable `i`, on écrit :

```
echo $i
```

et pour un changement de répertoire :

```
cd $mon_cv
```

Le caractère `$` est spécial et indique au shell d'effectuer une substitution de variable qui consiste à remplacer le nom de la variable qui suit le caractère `$` par sa valeur. Dans notre exemple, la variable `i` est remplacée par sa valeur puis la commande `echo` est exécutée avec la valeur `1` comme argument.

En plus de la particularité des espaces dans l'affectation, il y a une autre chose à retenir sur les variables du shell : l'accès à une variable sans affectation préalable ne produit pas d'erreur et retourne le valeur vide (*null value*). La commande `echo` ci-dessous (pour laquelle on suppose que la variable `qqchose` n'existe pas) montre qu'il n'y a rien qui a remplacé le nom de la variable. Les deux points sont affichés juxtaposés.

```
$ echo :$qqchose:
::
```

Pour initialiser une variable à la valeur vide, la bonne manière est de faire une affectation de la valeur vide exprimée par deux doubles quotes (`""`) de la façon suivante :

```
qqchose=""
```

C'est une bonne pratique de déclarer une variable par son initialisation avant son utilisation. De plus, si le Bash est configuré avec la commande `set -u`, le Bash détecte une erreur lorsqu'il y a accès à une variable non existante. L'erreur ne se produit pas si la variable est initialisée explicitement avec la valeur vide.
Notons que la commande interne `unset` sert à supprimer une variable. Pour supprimer la variable `qqchose`, entrons :

```
$ unset qqchose
$ set -u
$ echo $qqchose
-bash: qqchose: unbound variable
```

La commande `set -u` (*unset*) change le comportement du Bash lorsqu'il il y a accès d'une variable qui n'existe pas. Comme le montre l'exemple, une erreur se produit.

2.1 Concaténation de chaînes de caractères

La substitution de variable écrite sous la forme `$variable` est la forme simplifiée de l'écriture sous sa forme complète. La forme complète utilise des accolades : `${variable}`. La forme complète s'utilise quand le caractère qui suit le nom de la variable est un caractère alphanumérique ou le caractère souligné. Par exemple, si la variable `nombre` est initialisée avec la chaîne de caractères `10` de la manière suivante :

```
nombre=10
```

Afficher la chaîne de caractères `101` en utilisant la variable `nombre` se réalise avec la forme complète de la substitution de variable :

```
$ echo ${nombre}1
101
```

En l'absence des accolades, la substitution de variable s'applique sur la variable de nom `nombre1` qui elle n'existe pas. La valeur vide sera alors obtenue.

Cet exemple illustre la concaténation de chaînes de caractères. Le principe de la concaténation est assez simple, il suffit de juxtaposer les chaînes pour en former une nouvelle. Supposons que la variable `un` a été définie :

```
$ un=1
```

La concaténation de la chaîne `nombre` et de la chaîne `un` s'écrit donc :

```
$ echo $nombre$un
101
```

2.2 Manipulation de chaînes de caractères

La substitution de variable retourne la valeur de la variable dans le cas simple. Mais la substitution de variable offre de nombreuses autres possibilités à appliquer sur la valeur de la variable qui est retournée. Ainsi le Bash possède des fonctions pour manipuler les chaînes de caractères. Il est rappelé qu'une valeur dans une variable est de type chaîne de caractères.

Pour obtenir la longueur en nombre de caractères de la valeur d'une variable, nous allons écrire `${#variable}`.

```
$ var=debut-milieu-fin
$ echo ${#var}
16
```

Par défaut, une variable sans affectation préalable retourne le vide. Ceci peut poser des problèmes dans les scripts comme nous le verrons dans la séquence 4. Il y a deux façons de gérer l'accès à une variable lorsqu'elle n'est pas définie. La première façon est de réagir par une erreur d'exécution en changeant le comportement par défaut du Bash avec la commande `set -u`. La seconde façon est d'anticiper le manque de valeur et d'indiquer une valeur par défaut à retourner. Ceci se réalise avec la substitution de variable avec valeur par défaut et s'écrit `${variable:-valeur}`.

```
$ echo ${qqchose:-rien}
rien
```

La substitution de motifs appliquée sur la valeur de la variable est bien pratique pour changer la valeur retournée et s'écrit `${variable/motif/nouveau}`. La première occurrence du motif dans la valeur est changée par le nouveau motif. Les règles d'écriture d'un motif sont les mêmes que celles utilisées par la substitution de nom de fichier.

```
$ var=debut-milieu-fin
$ echo ${var/-/_}
debut_milieu-fin
```

Pour appliquer la substitution de motifs sur toutes les occurrences et non pas uniquement sur la première, il faut doubler le caractère barre oblique `'/'` de cette manière `${variable//motif/nouveau}`

Il y a aussi la substitution de variables avec suppression du préfixe ou du suffixe. L'écriture `${variable#motif}` supprime le préfixe correspondant au motif de la valeur. Si le caractère dièse `"#'` est doublé, cela signifie qu'il faut retenir le préfixe le plus grand. Prenons l'exemple d'un nom de fichier pour illustrer cette forme de substitution.

```
$ filename=/home/alice/fich.txt
$ echo ${filename#/*/}
alice/fich.txt
$ echo ${filename##/*/}
fich.txt
```

Pour supprimer le suffixe, l'écriture est `${variable%motif}`. Si le caractère pourcentage `"%'` est doublé comme pour le cas précédent, c'est la plus longue correspondance du suffixe qui est supprimée.

```
$ echo ${filename%/*}
/home/alice
```

Un moyen de retenir cette syntaxe est de regarder son clavier. Le caractère `'#'` est à gauche, comprendre "traite le préfixe" (le préfixe est devant le suffixe !). Quant au caractère `'%'`,il est placé à droite et donc il indique qu'il traite le suffixe.

Enfin terminons par l'extraction de sous-chaînes de la valeur de la variable, notée `${variable: position:longueu`
De la position indiquée, le nombre de caractères est extrait de la valeur contenue dans la variable.

```
$ echo ${var: 6:6}
milieu
$ echo ${var: 6}
milieu-fin
$ echo ${var: -3}
fin
```

Comme le montre l'exemple, si la longueur n'est pas indiquée, l'extraction est faite jusqu'à la fin de la chaîne de caractères. Si la position est un nombre négatif, la position commence en partant de la fin de la chaîne de caractères. Il est à noter que le signe négatif doit être précédé par un caractère espace pour ne pas être interprété comme une substitution de variables avec une valeur par défaut.
De manière évidente, `${variable: 0}` est bien sûr équivalent à `$variable`

3 Substitution de commande

La substitution de commande consiste à remplacer une commande par le résultat de cette commande. Cette substitution s'écrit à l'aide de la construction `$(`*command*`)` où le terme *command* est le nom de la commande qui doit être remplacée par le résultat de son exécution. Notez qu'il n'est pas utile de mettre une espace entre les parenthèses et la commande.
Anciennement, la substitution de commande était indiquée entre deux symboles ‘ (*back quote*). Moins lisible, cette écriture est devenue obsolète. Nous ne l'utiliserons pas et nous ne la citerons plus.

Séparateurs de mots

La séparation des mots dans une ligne de commande ne se fait pas uniquement que par les espaces. Le shell interprète aussi les caractères suivants comme des séparateurs de mots : tab, `'|'`, `'&'`, `';'`, `'('`, `')'`, `'<'`, `'>'`. Cela implique qu'il est inutile de placer des espaces autour de ces caractères.

La substitution de commande s'utilise pour donner comme argument à une commande le résultat d'une autre commande. Par exemple une ligne de commande qui affiche un message suivi du chemin d'accès du répertoire courant peut s'écrire de la manière suivante :

```
$ cd
$ echo ici $(pwd)
ici /home/alice
```

La commande `pwd` est exécutée en premier et elle est remplacée par son résultat. Puis la commande `echo` est exécutée avec deux arguments : la chaîne de caractères `ici` et le résultat de la commande `pwd`.

La substitution de commande peut aussi s'utiliser avec l'affectation pour indiquer une valeur.

```
$ rep=$(pwd)
$ echo $rep
/home/alice
```

Dans notre exemple, la variable `rep` contient le chemin d'accès au répertoire courant.

On peut noter qu'il n'y a pas de contrainte sur le résultat produit par la substitution de commande. Les commandes produisant des résultats sur plusieurs lignes peuvent elles aussi être substituées. L'exemple suivant affecte comme valeur à une variable le contenu d'un fichier.

```
names=$(cat lipsum.txt)
```

Exercice 2.3.1: Alice veut y voir plus clair

Alice veut y voir plus clair dans les constructions syntaxiques. Elle définit la variable `dat` contenant la chaîne `pwd`. Avec comme répertoire courant, le répertoire `/home/alice/Compagnon`, elle entre les commandes suivantes :

1. `$dat`
2. `echo dat`
3. `echo $dat`
4. `echo ${dat}`
5. `echo $(dat)`
6. `echo ${$dat}`
7. `echo $($dat)`
8. `echo $data`
9. `echo ${dat}a`

1/ Commencer par affecter la chaîne `pwd` à la variable `dat`. Quelle est cette ligne de commande ?
2/ Pour chaque commande de Alice, indiquer le résultat affiché.

Solution page 110

4 Inhibitions

Il y a un moyen de contrôler l'interprétation des caractères spéciaux et des substitutions faites par le shell. C'est ce que nous appelons les inhibitions (*quoting*). Nous distinguons 3 formes d'inhibitions comme l'indique le tableau 1.

\	inhibition d'un caractère, le caractère suivant est pris littéralement.
" "	inhibition partielle : la substitution de variable, la substitution de commande et l'inhibition caractère sont maintenues.
' '	inhibition totale, la chaîne de caractères est prise littéralement.

Tableau 1 – Les inhibitions du shell.

4.1 Inhibition totale

L'inhibition totale empêche toutes les interprétations. Elle est notée entre deux simples quotes (aussi appelés apostrophes). Elle sert à composer une chaîne de caractères contenant des caractères spéciaux. Ainsi la chaîne de caractères délimitée entre deux simples quotes est prise de manière littérale. Par exemple, l'affichage de la chaîne de caractères `* $(pwd)` est obtenu par la commande `echo '* $(pwd)'` dans laquelle la chaîne de caractères est indiquée avec une inhibition totale. Aucune interprétation n'est effectuée.

4.2 Inhibition partielle

L'inhibition partielle, notée entre deux doubles quotes (aussi appelés guillemets), autorise les substitutions de variables, les substitutions de commandes et l'interprétation du caractère spécial anti-slash '\'. L'expansion des caractères spéciaux pour les noms des fichiers et l'espace comme séparateur de mots ne sont plus opérants. Prenons l'exemple d'une variable contenant le caractère spécial `*` qui dans l'interprétation des noms de fichiers est remplacé par les noms de tous les fichiers du répertoire courant.

```
a=*
```

On peut déjà noter que l'affectation s'effectue sans interpréter les caractères spéciaux du nom de fichier. Ceci se vérifie en affichant le contenu de la variable `a` en inhibition partielle. Dans l'exemple ci-dessous, la substitution de la variable `a` est appliquée mais pas l'interprétation des noms de fichier car il s'affiche le caractère `*`.

```
$ echo "$a"
*
```

Pour voir la différence, si on recommence cette commande sans l'inhibition partielle de l'argument de la commande `echo`, on obtient comme affichage le contenu du répertoire courant. Ceci montre que le caractère `*` a été interprété lorsqu'il a été argument de la commande `echo`.

Intéressons-nous aux espaces. Les espaces sont utilisées par le shell comme séparateur de mots. Lorsqu'elles sont utilisées à l'intérieur des doubles quotes (et de surcroît à l'intérieur des simples quotes), elles ne sont plus interprétées mais prises littéralement. Ainsi le nombre d'espaces est conservé.

Dans l'exemple, ci-dessous nous affichons deux espaces entre deux caractères ' :'

```
$ echo  :  :
: :
```

L'affichage ne comporte plus qu'une seule espace entre les deux caractères. La commande `echo` a été appelée avec deux arguments. Si nous recommençons cet affichage mais avec l'inhibition partielle, le nombre d'espaces n'est pas modifié.

```
$ echo ":  :"
:  :
```

L'explication est que la commande `echo` a été appelée avec un seul argument qui est la chaîne de caractères comprise entre les doubles quotes. C'est donc l'affichage de cette chaîne de caractères qui est effectué.

L'usage principale de l'inhibition partielle porte sur la composition de chaînes de caractères. Par exemple, deux variables peuvent se combiner pour composer une chaînes de caractères. Supposons :

```
$ a=Bash
$ b=genial
$ phrase="$a est $b"
$ echo "$phrase"
Bash est genial
```

La variable `phrase` est formée par l'insertion du contenu de 2 variables. Les doubles quotes sont obligatoires dans la commande d'affectation pour inhiber la fonction de séparateur de mots effectuée par l'espace.

Pour terminer sur cette section sur les doubles quotes, il faut souligner quelques bonnes pratiques d'utilisation :

— Les arguments de la commande `echo` sont quasiment toujours à encadrer par des doubles quotes (sauf si une substitution de nom de fichier doit être utilisé). C'est une assurance de voir l'affichage s'effectuer comme il a été écrit et donc d'éviter les surprises. Prenons l'exemple de l'affichage sur 2 lignes. Le saut de ligne s'indique par le caractère composé `'\n'` et l'interprétation des caractères composés par l'option `-e` de la commande `echo`.

```
$ echo -e la date est : \n $(date)
la date est : n Web Feb 10 14:22:18 GMT 2021
$ echo -e "la date est : \n $(date)"
la date est :
 Web Feb 10 14:22:24 GMT 2021
```

Lorsque l'on ne maîtrise pas toutes les subtilités de l'interpréteur shell, cet exemple montre que l'encadrement d'une chaîne de caractères par des doubles quotes est un bon moyen d'avoir un contrôle de ce qui sera affiché.

— Vous le verrez dans la séquence 4 pour les scripts, lorsque qu'un argument est attendu et que cet argument est rendu par une substitution de variable. Si la variable contient la valeur vide alors il n'y a rien (le vide) à la place de l'argument. C'est une cause d'erreur très classique. Utilisons la commande `set --` pour traiter les arguments d'une commande et `$#` la variable du comptage d'arguments [1] :

```
$ qqchose=""
$ set -- $qqchose ; echo $#
0
$ set -- "$qqchose" ; echo $#
1
```

Dans le premier cas, rien n'est rendu par la substitution de la variable `qqchose`. Dans le second cas, la substitution de la variable `qqchose` ne rend toujours rien mais dans le cas d'une chaîne de caractères, cela prend la signification valeur vide (comme nous l'avons vu à la section 2) et notée `""` .

1. Le point virgule est un séparateur de commande

4.3 Inhibition de caractère

La dernière forme d'inhibition est l'inhibition de caractère (aussi appelée échappement). Elle se note avec le caractère anti-slash '\' préfixant le caractère à inhiber. Comme son nom l'indique, elle empêche l'interprétation d'un caractère spécial du shell. Par exemple, pour afficher `$a` littéralement, il faut empêcher la substitution de variable et donc inhiber l'interprétation du caractère `$` comme on peut le voir ci-dessous :

```
$ echo \$a
$a
```

L'inhibition de l'espace s'utilise pour supprimer sa fonction de séparateur de mots. On peut en avoir besoin pour maintenir le nombre d'espaces entre deux mots. Par exemple pour afficher deux espaces entre les deux points, on va écrire :

```
$ echo : \ :
:  :
```

L'inhibition de la fin de ligne empêche la terminaison de la ligne de commande. Celle-ci peut se poursuivre sur la ligne suivante. Vous remarquerez que l'invite de commande a changé après la première ligne.

```
$ a="une \
> deux"
$ echo $a
une deux
```

Ainsi le caractère `'\'` en fin de ligne est traité comme une indication de continuation de ligne.

✍ **Challenge C23Q1**

Noël est passé

4.4 Ordre de priorité

Les différentes formes d'inhibitions peuvent s'imbriquer. Le tableau 2 indique pour l'inhibition totale (première ligne) et l'inhibition partielle (seconde ligne) les caractères spéciaux interprétés, autrement dit les substitutions ou les inhibitions appliquées. Le caractère `'$'` symbolise la substitution de variable et de commande. Le terme glob indique la substitution de nom de fichier.

	'	"	\	$	glob
'	f	-	-	-	-
"	-	f	i	i	-

Tableau 2 – Légende : i interprété, - non interprété, f quote de fin.

Ainsi dans une inhibition partielle, la simple quote (') perd sa signification spéciale. Autrement dit, il n'y a pas de substitution totale dans une substitution partielle. Par exemple, l'affichage du mot `Aujourd'hui` s'écrit par la ligne de commande :

```
echo "Aujourd'hui : $(date)"
```

La simple quote est prise ici littéralement et a perdu sa signification spéciale.

✍ **Challenge C23Q2**

Ce n'est pas la croix et la bannière

✍ **Challenge C23Q3**

Bon ou mauvais

Le shell procède à l'interprétation de la ligne de commande en effectuant les opérations dans l'ordre suivant :

1. substitution de variable
2. substitution de commande
3. substitution de noms de fichier.

Pour effectuer les substitutions dans un ordre différent ou pour appliquer deux fois de suite le même type de substitution, la commande interne `eval` propose de résoudre ce problème. Cette commande relance la phase d'interprétation du shell. Le résultat de la première interprétation sert à constituer la ligne de commande pour la seconde interprétation. C'est le résultat de cette interprétation qui sera exécuté. Prenons l'exemple d'une redirection. La variable `x` contient la chaîne de caractères "100". Puis nous définissons la variable `ptr` qui contient le nom de la variable `x`.

```
$ x=100
$ ptr=x
```

Nous voulons afficher le contenu de la variable dont le nom est contenu dans la variable `ptr`. La commande `echo $ptr` affichera 'x'. La commande `echo \$$ptr` affichera '$x'. Pour que ce dernier résultat soit de nouveau interprété, nous devons utiliser la commande `eval` :

```
$ echo $ptr
x
$ echo \$$ptr
$x
$ eval echo \$$ptr
100
```

Le résultat affiché est bien 100 maintenant.

Pour illustrer un ordre différent, prenons un exemple dans lequel la substitution de commande est effectuée avant la substitution de variable :

```
$ cmd=pwd
$ \$$(echo cmd)
-bash: $cmd: command not found
$ eval \$$(echo cmd)
/home/alice
```

Dans cet exemple, la substitution de variable est inhibée afin que la première évaluation effectue uniquement la substitution de commande. Au terme de cette évaluation, il reste `$cmd` à évaluer. La commande `eval` effectue la substitution de variable et appelle la commande `pwd` qui était contenue dans la variable `cmd`. On peut noter que c'est un exemple qui n'a pas beaucoup de sens car on obtient le même résultat avec la ligne de commande :

```
$ $cmd
/home/alice
```

Exercice 2.3.2: Alice y voit plus clair

1/ Dans l'exercice précédent, Alice était tombée sur une erreur lorsqu'elle avait demandé une double substitution de variable.

```
$ echo ${$dat}
${$dat}: bad substitution
```

Supposons que `dat=HOME`, corriger la ligne de commande en erreur pour que la double substitution de variable fonctionne.

2/ Elle commence à y voir plus clair entre les substitutions et les inhibitions. Elle décide de reprendre également la dernière ligne de commande de l'exercice précédent :

```
$ dat=pwd ; pwda=bravo
$ echo ${dat}a
pwda
```

Compléter la ligne de commande pour afficher le contenu de la variable dont le nom est la concaténation du contenu de la variable `dat` et de la lettre `'a'`.

3/ Alice veut afficher la ligne de commande avant de l'exécuter. Comme exemple, elle retient l'affichage de la variable `fox` définie comme `fox=bravo`. Pour cela, elle définit la variable `cmdline` contenant la ligne de commande prise en exemple. Puis elle affiche le contenu de `cmdline` et enfin elle exécute le contenu de la variable `cmdline`. Donner les 3 lignes de commandes que doit entrer Alice pour faire cet exercice.

Solution page 111

✍ **Challenge C23Q4**

Variable mystère

5 Conclusion

Dans cette activité vous avez appris les constructions syntaxiques utilisées dans une ligne de commande. Ces constructions sont traitées par le shell lors de la phase d'interprétation, qui est effectuée après la validation de la ligne de commande.

Ainsi l'interprétation de la ligne de commande :

- commence par les substitutions de variable,
- continue par les substitutions de commande
- et se termine par les substitutions des caractères spéciaux pour les noms de fichiers.

Vous avez vu aussi comment contrôler cette interprétation et son ordre avec les inhibitions et la commande `eval`.

Vous pouvez maintenant comprendre pourquoi on parle de ligne de commande. La ligne de commande, à l'image d'une phrase, comporte des mots qui se combinent. Certains de ses mots sont définis par les constructions que nous venons de voir.

Dans l'activité suivante, changement de programme ! Après avoir vu la phase d'interprétation de la ligne de commande, vous allez voir comment sont gérées les entrées et les sorties des commandes lors de la phase d'exécution.

Solutions des exercices

Solution de l'exercice 2.3.1 page 103:
Il faut commencer par définir la variable de cette manière :

```
dat=pwd
```

L'exécution des commandes de Alice donne ce résultat :

```
$ $dat ; # 1
/home/alice/Compagnon
$ echo dat ; # 2
dat
$ echo $dat. ; # 3
pwd
$ echo ${dat} ; # 4
pwd
$ echo $(dat) ; # 5
dat
dat: command not found
$ echo ${$dat} ;  # 6
${$dat}: bad substitution
$ echo $($dat) ; # 7
$dat
/home/alice/Compagnon
$ echo $data ; # 8

$ echo ${dat}a ; # 9
pwda
```

Les explications sont, pour chaque commande :

1. exécution de la commande `pwd` obtenue après substitution de la variable.
2. affichage de la chaîne `dat`.
3. affichage du contenu de la variable.
4. même résultat que précédemment par utilisation de la forme complète de la substitution de variable.
5. substitution de la commande `dat` qui n'existe pas.
6. substitution de variable dans la forme complète de la substitution de variable. Le même type de substitution est à appliquer séquentiellement. Cela ne marche pas car l'interpréteur applique le même type de substitution en même temps (simultanément).
7. substitution de variable dans une substitution de commande. Comme cela fonctionne, on en déduit que la substitution de variable est effectuée avant la substitution de commande.
8. substitution de la variable `data` qui n'existe pas. Bien que la variable n'existe pas, aucune erreur n'est notifiée. Ce comportement du Bash peut être changé par la commande `set -u` comme nous l'avons vu précédemment.

   ```
   $ set -u ; echo $data
   -bash: data: unbound variable
   ```

9. avec la forme complète de la substitution de variable, c'est le contenu de la variable `dat` qui est concaténé avec la lettre `'a'`.

Solution de l'exercice 2.3.2 page 108:
1/ Il faut faire deux substitutions de variable l'une après l'autre. Il faut indiquer au shell que le premier caractère `'$'` est à prendre littéralement. L'inhibition caractère sert à cela, on écrit donc :

```
$ echo \${$dat}
${HOME}
```

Ensuite il faut refaire une autre évaluation pour substituer la variable `HOME` avant de faire l'exécution de la ligne de commande :

```
$ eval echo \${$dat}
/home/alice
```

2/ le résultat de la première évaluation donne le nom de la variable, il faut donc faire une seconde évaluation en faisant une substitution de variable sur le nom obtenu. Pour cela il faut inhiber la caractère `'$'` lors de la première évaluation. Il doit perdre son caractère spécial pour cette évaluation.

```
$ eval echo \$${dat}a
bravo
```

Note : cette opération en programmation s'appelle une indirection. Une variable référence une autre variable qui contient une valeur. La référence dans le cas du Bash se présente sous la forme du nom de la variable qui contient la valeur. Le Bash propose l'écriture `${!var}` pour faire la substitution de la variable `var` avec une indirection. Dans notre exercice, ceci deviendrait

```
$ var=${dat}a  # soit var=pwda
$ echo ${!var}
bravo
```

3/ Les trois lignes sont les suivantes :

```
$ cmdline='echo "$fox"'
$ echo $cmdline
echo "$fox"
$ eval $cmdline
bravo
```

La première ligne définit la variable `cmdline` qui contient la ligne de commande en prenant soin d'inhiber en totalité cette ligne de commande. La seconde ligne affiche la ligne de commande. La variable `fox` n'est pas substituée car elle est contenue dans la variable `cmdline` qui est substituée. Il faut une seconde évaluation. C'est ce que fait la troisième ligne qui fait deux évaluations avant d'exécuter la commande.

Activité 2.4

Contrôler l'exécution des commandes

1 Introduction

Une fois l'analyse de la ligne de commande effectuée comme nous venons de la voir dans l'activité précédente, le shell va maintenant exécuter la commande.

Le shell a la possibilité, soit d'exécuter lui-même la commande demandée par l'utilisateur, soit de lancer l'exécution du programme correspondant. Dans le premier cas, on parle de commande interne et dans le second cas de commande externe. À une commande externe correspond un fichier exécutable ayant comme nom, le nom de la commande. L'exécution d'une commande externe entraîne la création d'un nouveau processus exécutant le programme correspondant à la commande. Au cours de cette activité, nous allons apprendre à contrôler l'exécution des commandes et définir la notion de processus. Avec cette notion, nous rentrons dans le cœur du fonctionnement d'un système d'exploitation.

2 Notion de processus

2.1 Qu'est-ce qu'un processus ?

Une commande peut se définir comme étant un algorithme. Cet algorithme peut être sous la forme d'un fichier contenant un programme prêt à être exécuté par l'ordinateur. Un processus est alors un programme en cours d'exécution. Un programme donné n'existe généralement qu'en un exemplaire dans un ordinateur, mais il peut donner naissance à un nombre indéterminé de processus qui l'exécutent en même temps. Plus précisément, un processus correspond donc à une instance d'une commande en exécution. L'activité du processus de sa vie à sa mort est gérée par le système d'exploitation. Le processus est constitué par le code du programme et par les données comme les variables qu'il manipule. De plus il est décrit dans le système par un contexte d'exécution dans lequel sont pris en compte l'état d'avancement de l'exécution, les ressources utilisées, les droits associés et les attributs du processus. Aussi, comme un processus est une instance d'une commande en exécution, un système d'exploitation gère plusieurs processus en même temps.

2.2 Vie et mort naturelle des processus

La création d'un processus s'effectue par un autre processus. Le shell Bash est lui-même un processus avec lequel l'utilisateur interagit. Lorsqu'il valide une ligne de commande par un retour à la ligne, les traitements syntaxiques sont effectués. C'est ce que nous avons vu dans l'activité précédente. Ensuite, le shell invoque un programme particulier. Il recherche ce programme dans un des répertoires indiqués par la variable d'environnement `PATH`. Quand le fichier du programme est trouvé, le shell crée un clone

de lui-même (connu sous le nom de sous-shell) et demande au système d'exploitation de remplacer le sous-shell par le programme contenu dans le fichier. Le programme, sous la forme d'un processus, peut alors commencer son exécution. Un processus meurt quand il n'a plus de traitement à effectuer. Dans certains cas, un processus peut aussi mourir suite à une interruption. On notera que le processus créé est qualifié de processus fils et que le processus créateur se nomme le processus parent (ou père). La figure 2.4.1 représente le cycle de vie d'une commande sous la forme d'un processus. Le rôle principal du père est d'attendre la fin du fils. Le fils doit faire `exit()` pour se terminer. À la création, le fils remplace le code et les données du père mais conserve les fichiers ouverts. L'avantage pour le père de ne pas exécuter la commande lui-même est qu'en cas d'erreur, il n'est pas affecté et peut contrôler l'erreur. À noter que quand le père meurt, il « tue » tous ses fils.

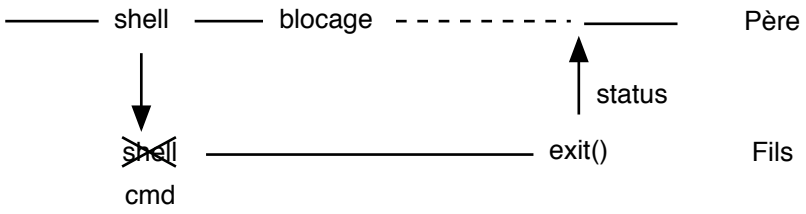


FIGURE 2.4.1 – Cycle de vie d'une commande.

Premier processus

Comme un processus est créé par un autre processus, se pose alors la question : qui a créé le premier processus ? La réponse se trouve dans la procédure de démarrage d'un ordinateur. Le système d'exploitation, en fin de procédure de démarrage, lance le processus init qui a pour but d'exécuter les différents scripts de configuration des services ou des applications de l'ordinateur. Le processus init n'a pas de parent.

Quand le processus se termine, son état de fin se matérialise par un code retour qui est disponible au processus parent. Par ce code, il indique au processus parent comment il s'est terminé. La notation pour indiquer le code retour est `'$?'`. Illustrons cela par un exemple : on exécute la commande `ls fichier1.txt` puis on affiche le code retour à l'aide de la commande `echo` de la manière suivante :

```
$ ls fichier1.txt
$ echo $?
0
```

Si la valeur retournée vaut zéro, cela indique un bon déroulement de la commande. Le processus s'est terminé correctement. Reprenons la manipulation précédente mais cette fois avec un fichier qui n'existe pas :

```
$ ls fichier3.txt
ls: fichier3.txt: No such file or directory
$ echo $?
1
```

Le message d'erreur affiché nous précise que le fichier n'existe pas. L'affichage du code retour donne une valeur différente de zéro. Par convention, un code retour non nul indique une situation anormale. Autrement dit, l'exécution de la commande ne s'est pas déroulée comme attendu et a rencontré une erreur.

Challenge C24Q1

Aller - retour

2.3 Principaux attributs d'un processus

Les informations relatives à un processus sont enregistrées dans un contexte au moment de sa création. Certaines de ces informations vont changer au cours du temps. Ce contexte comprend, entre autres, les attributs qui caractérisent le processus comme :

- PID (*Process ID*) ; c'est le numéro du processus qui l'identifie. Il est attribué à la création par le système d'exploitation.
- PPID (*Parent PID*) ; c'est le PID du processus parent qui a créé ce processus,
- UID (*User ID*) ; l'identifiant de l'utilisateur qui a lancé le processus. En d'autres termes, c'est le nom du propriétaire du processus.
- TTY correspond au terminal d'attachement du processus. C'est le terminal à partir duquel a été créé le processus.
- Sa priorité (système (PRI)) est donnée par l'usager (NI).
- STIME (*Start Time*) ; l'heure du démarrage du processus.
- TIME indique le temps cumulé de la CPU (*Central Processing Unit*) du processus. Il s'agit du temps de calcul pris. À noter qu'avec la Weblinux, cet attribut n'a pas une valeur valide.
- enfin, CMD (*CoMmanD*) indique la commande et ses arguments exécutée par ce processus.

Priorité de processus

La priorité d'un processus sert au système d'exploitation à ordonnancer l'exécution des processus actifs. Cette priorité dépend de la priorité attribuée par le système d'exploitation à la création du processus et de celle attribuée par l'utilisateur. Plus la valeur de priorité est faible, plus le processus est prioritaire. Un processus prioritaire s'exécutera plus rapidement.

La commande pour afficher les informations relatives aux processus se nomme `ps` (*Process State*). Par défaut, `ps` affiche tous les processus de l'utilisateur associés au même terminal.

```
 PID TTY          TIME CMD
  84 ttyS0    15049-12:58:15 bash
 141 ttyS0    15049-12:58:15 sleep
 143 ttyS0    15049-12:58:15 yes
 146 ttyS0    15049-12:58:15 ps
```

La commande `ps` possède de nombreuses options pour consulter les processus en cours dans votre système. Comme `ps` est une commande qui se veut compatible à plusieurs spécifications Unix, elle supporte le format d'option avec un tiret, sans tiret et avec double tiret. Comme beaucoup de commandes, les options disponibles sont affichées avec l'option `--help`. On distingue les options qui traitent du format de sortie de celles qui posent des critères de sélection des processus. Parmi les options de sélection, citons :

- `-e` tous les processus en cours de tous les utilisateurs,
- `x` les processus non attachés à un terminal.

Dans les options pour le format de sortie, on peut retenir :

- `-f` pour un format dit détaillé (*full-format*),
- `-l` pour un format long pour avoir l'état des processus et d'autres informations liées à la priorité du processus,
- `u` pour les informations d'usage,
- `-o` avec un format défini par l'utilisateur. La liste des indicateurs de format sont obtenus par l'option `L`.

Pour avoir plus de détails sur ces processus, l'utilisateur peut utiliser l'option `-f`. Cette option affiche quatre informations supplémentaires dont le UID, le PPID et le STIME.

```
UID        PID  PPID  C STIME TTY          TIME CMD
alice       84     1  0  1902 ttyS0    15049-12:58:15 -bash
```

```
alice      141    84  0  1902 ttyS0    15049-12:58:15 sleep 1000
alice      143    84  0  1902 ttyS0    15049-12:58:15 yes
alice      147    84  0  1902 ttyS0    15049-12:58:15 ps -f
```

L'option `u` affiche l'état du processus et les ressources de la machine consommées.

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice       83  0.0  7.7   2656  2016 ttyS0    Ss+   1902 21671338:15 -bash
alice       86  0.0  1.6   2256   424 ttyS0    S+    1902 21671338:15 sleep 1000
alice       87  0.0  1.5   2256   392 ttyS0    R+    1902 21671338:15 yes
alice       91  0.0  2.5   1432   656 ttyS0    R+    1902 21671338:15 ps u
```

Un processus peut passer par plusieurs états lors de son exécution. L'état indique l'activité du processus. On distingue l'état « en sommeil » (*Sleeping*), noté S, pour un processus en attente d'un événement. L'état « bloqué » (*Disk sleep*), noté D, est presque similaire à l'état S mais concerne les opérations d'entrée-sortie. L'état « actif » (*Running*), noté R, correspond à un processus en traitement par la CPU. L'état « arrêté » (*sTopped*), noté T, indique que l'exécution du processus a été suspendue temporairement. Elle pourra redémarrer sur demande explicite. Si aucun format ne vous convient, il vous reste à spécifier votre format avec l'option `-o` :

```
$ ps -o user,s,pid,pcpu,cmd
USER     S   PID %CPU CMD
alice    S    84  0.0 -bash
alice    S   141  0.0 sleep 1000
alice    R   143  0.0 yes
alice    R   151  0.0 ps -o user,s,pid,pcpu,cmd
```

Enfin, pour une mise à jour en temps réel de la sélection et de l'affichage, l'utilisation de la commande `top` est préférable.

Pour évaluer la performance de l'exécution d'une commande, la commande `time` retourne les statistiques temporelles en fin d'exécution. Elle doit précéder la commande à évaluer comme par exemple :

```
$ /usr/bin/time sleep 10
real  0m 10.05s
user  0m 0.00s
sys     0m 0.02s
```

2.4 Processus en action

Maintenant que nous savons comment un programme est traité par le système d'exploitation, nous allons voir comment tirer pleinement profit de la notion de multi-tâches offerte par les systèmes de type Unix. Quand vous ouvrez une session dans une console, vous avez accès au shell Bash pour exécuter des commandes. Par défaut, le shell s'utilise en effectuant des tâches séquentiellement. Une tâche est par exemple une commande à exécuter. Lorsque l'utilisateur lance une commande, il doit attendre la fin de la commande en cours pour lancer la commande suivante.
Si vous voulez exécuter deux commandes en même temps comme par exemple, lire le manuel d'une commande et effectuer la saisie de la commande, avec la console comme unique terminal, vous allez vous dire que ça n'est pas pratique. Pas d'inquiétude, il y a des solutions comme le multiplexer de terminal `tmux`[1] (*terminal multiplexer*). Cette commande sert à exploiter plusieurs terminaux virtuels au sein d'une seule et même console.

1. `tmux` n'est pas mis en œuvre dans la Weblinux

Environnement graphique

Dans les environnements graphiques, `tmux` est moins pertinent. Dans ces environnements, il est facile d'ouvrir plusieurs fenêtres avec un terminal dans chacune.

Une fois la commande `tmux` saisie, l'utilisateur ouvre une session tmux dans laquelle il peut créer autant de terminaux virtuel qu'il a besoin. Les terminaux sont organisés en une liste circulaire. Changer de terminal consiste à activer le terminal suivant (ou précédent) dans la liste. Les commandes internes de `tmux` sont envoyées par des combinaisons de touches. Les combinaisons de touches les plus courantes sont :

CTRL + b puis c	Créer un nouveau terminal dans la console.
CTRL + b puis n	Passer au terminal suivant dans la liste (*next*).
CTRL + b puis w	Afficher la liste des terminaux.
CTRL + b puis &	Supprimer le terminal virtuel courant.

Outre la commande `tmux`, le shell Bash offre un moyen pour un fonctionnement en multi-tâches. C'est ce que nous vous proposons d'étudier maintenant avec la notion de tâche d'exécution (*job*).

3 Les tâches d'exécutions

Lorsque qu'un programme n'a pas besoin d'interagir avec le terminal, il est alors judicieux d'exécuter ce programme comme un processus d'arrière plan. En arrière plan, le terminal n'est alors pas bloqué par l'exécution de ce processus. Le shell n'attend pas la fin de l'exécution du processus mais affiche de nouveau l'invite de commande pour lancer une autre commande. A contrario, les commandes lancées qui bloquent le shell pendant leur exécution sont dites en avant plan. Avec la notion d'arrière plan, l'utilisateur peut alors lancer des commandes qui s'exécutent en parallèle. Cette fonctionnalité de gestion des processus (*Job control*) attachés au shell est très importante. D'ailleurs, on ne parle plus de processus mais de tâches d'exécutions pour ces processus sous le contrôle de l'utilisateur par son shell. Une tâche, c'est même plus qu'un processus. Elle peut être constituée par un groupe de processus, comme cela peut être le cas dans des commandes qui s'enchaînent sur une même ligne de commande.

La grande question maintenant est : Mais comment faisons-nous cela ?
Pour les besoins de la démonstration, nous utiliserons la commande `sleep` qui consiste à suspendre l'exécution pour une durée déterminée, autrement dit à créer un processus qui a une durée de vie au moins égale à la durée indiquée. Ainsi la commande `sleep 10` bloque le terminal pour une durée de 10 secondes puis la commande se termine.

Pour ne pas être bloqué, nous pouvons lancer la commande `sleep` en arrière plan (*background*). Autrement dit, il faut lancer la commande en mode multi-tâches. Ceci s'effectue en ajoutant le caractère `&` (« et commercial » ou esperluette) à la fin de la commande. Recommençons la commande précédente mais en arrière plan cette fois-ci.

```
sleep 10 &
```

Commande interne

Les commandes internes sont des commandes du shell lui-même. Elles ne créent pas un nouveau processus. Leur description est dans le manuel du Bash.

Après la validation de la ligne de commande, le numéro de tâche s'affiche entre crochets suivi par le numéro de processus puis l'invite de commande s'affiche immédiatement. La commande `sleep`

s'exécute en arrière plan. Le terminal est disponible pour une autre commande. Justement, listons les tâches d'arrière plan qui sont en cours d'exécution attachées au shell. Pour cela nous disposons de la commande interne au shell `jobs`.

```
$ sleep 10 &
[1] 96
$ jobs
[1]+   Running            sleep 10 &
```

La seule tâche en cours d'exécution est notre commande `sleep 10 &`. Cette dernière est identifiée par le numéro de tâche 1. Le numéro de tâche est géré par le shell et non par le système d'exploitation comme le numéro de processus. Le numéro de tâche est un numéro de séquence. Cela signifie que le numéro d'une nouvelle tâche est l'incrémentation du numéro de tâche de la dernière tâche active. Exemple, le shell a les tâches 3 et 4 actives, la tâche 4 s'arrête. Une nouvelle tâche est activée, elle portera le numéro 4. Comme le montre cet exemple, le numéro de tâche n'est pas un identifiant mais un numéro de séquence.

```
$ jobs
[1]   Running                 sleep 600 &
[2]   Running                 sleep 700 &
[3]-  Running                 sleep 800 &
[4]+  Done                    sleep 10
$ jobs
[1]   Running                 sleep 600 &
[2]-  Running                 sleep 700 &
[3]+  Running                 sleep 800 &
$ sleep 5 &
[4] 152
$ jobs
[1]   Running                 sleep 600 &
[2]   Running                 sleep 700 &
[3]-  Running                 sleep 800 &
[4]+  Running                 sleep 5 &
```

Pour la ramener en avant plan (ou *foreground*), à savoir la faire quitter le mode multi-tâches, nous effectuons la commande interne `fg %1` avec le caractère `'%'` pour indiquer une référence à un numéro de tâche. La commande `sleep` bloque de nouveau le terminal.
Pour envoyer une commande en arrière plan, c'est-à-dire la mettre en exécution en mode multi-tâches, alors qu'elle est en cours d'exécution, il y a deux manipulations à faire :

1. la première, c'est de suspendre la commande en faisant la combinaison de touches CTRL+z,
2. la seconde, c'est de l'envoyer en arrière plan avec la commande interne `bg` pour *background*.

Nous pouvons vérifier que nous avons renvoyé la commande `sleep` en arrière plan, en faisant de nouveau la commande `jobs`. Vous venez de découvrir comment exécuter des commandes en parallèle.

4 Contrôle d'exécution

Il est important de savoir suspendre ou arrêter un processus lorsque celui-ci ne se comporte pas comme il le devrait. Le contrôle de l'exécution des processus s'effectue en envoyant des signaux. Un signal est un ordre donné à un processus. Un signal est envoyé par la commande `kill` dont la syntaxe est la suivante :

`kill [-`*nom du signal*|*numéro du signal*`]` *PID* ⋯

Ainsi un signal peut être donné par son nom ou son numéro. La liste des signaux disponibles est donnée par la commande `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
```

Cette liste indique le numéro associé à chaque nom de signal. Le nom a une forme abrégée en supprimant le préfixe SIG. Par exemple, le signal SIGTERM est identifié par le numéro 15 ou par le nom abrégé TERM. Dans cette liste de 31 signaux, nous pouvons souligner ceux qui peuvent vous être utiles, comme :

- SIGTERM pour terminer un processus avec fermeture normale de ses fichiers. Ce signal peut être ignoré par certain processus. C'est le signal par défaut de la commande `kill`.
- SIGKILL pour terminer brutalement et immédiatement un processus en toutes circonstances.
- SIGINT pour interrompre un processus.
- SIGQUIT pour interrompre un processus en créant une image du processus à l'instant de l'interruption. Cette image peut être utilisable par la suite avec une application de débuggage.
- SIGSTOP pour stopper ou arrêter provisoirement un processus.
- SIGCONT pour remettre en exécution un processus stoppé.
- SIGHUP (*hangup*) pour demander à un processus démarré par le système de relire ses fichiers de configuration.

Par défaut, la commande `kill` envoie le signal SIGTERM aux processus dont le PID est donné en argument. Si le signal SIGTERM est inopérant, il reste le signal SIGKILL pour supprimer à coup sûr le processus. Dans ce dernier cas, il y a un risque de perte de données non enregistrées sur le système de stockage. Pour arrêter le processus créé avec la commande `sleep` de l'exemple précédent, l'envoi du signal de terminaison de numéro 15 par la commande `kill` ou plus simplement `kill 96` va revenir à le tuer. La commande avec l'indication explicite du signal s'écrit donc :

```
$ kill -15 96
```

À la prochaine ligne de commande (une ligne vide par exemple), un message confirme la mort du processus de la commande `sleep`. Lorsque c'est une tâche qui doit être interrompue, le numéro de tâche remplace le PID comme argument de la commande `kill`. Si la commande `jobs` retourne une liste de trois tâches numérotées en séquence, arrêter la seconde tâche s'indique par la commande `kill %2`

Exercice 2.4.1: Ça envoie du signal

1/ Alice souhaite voir sa table des processus. Quelle est la commande qui génère cette sortie ? (La ligne de la commande ayant généré cette sortie a été supprimée pour l'exercice.)

```
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice     86  0.0  8.0   2728  2096 ttyS0    Ss    1903 21671338:15 -bash
alice    103  0.0  1.6   2256   424 ttyS0    S     1903 21671338:15 sleep 150
alice    104  0.0  1.6   2256   424 ttyS0    S     1903 21671338:15 sleep 250
```

2/ Le processus de PID 103 a été suspendu temporairement. Il est dans l'état arrêté comme le montre la sortie ci-dessous. Indiquer la commande de changement d'état de ce processus.

```
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice     86  0.0  8.0   2728  2096 ttyS0    Ss    1903 21671338:15 -bash
```

```
alice    103  0.0  1.6   2256   424 ttyS0    T      1903 21671338:15 sleep 150
alice    104  0.0  1.6   2256   424 ttyS0    S      1903 21671338:15 sleep 250
```

3/ Alice consulte la liste des tâches d'exécutions et obtient la liste ci-dessous. Donner la ligne de commande de lancement des deux tâches et la commande `kill` de suspension de la tâche 1. Quelle est la commande pour afficher la liste ?

```
[1]+  Stopped                 sleep 150
[2]-  Running                 sleep 250 &
```

4/ Donner la commande pour changer l'état de la tâche 2 de cette façon :

```
[1]+  Stopped                 sleep 150
[2]-  Terminated              sleep 250
```

5/ Donner la commande pour changer l'état de la tâche 1 de cette façon :

```
[1]+  Running                 sleep 150 &
```

puis de la passer en premier plan.

Solution page 122

Lorsque le PID n'est pas connu de l'utilisateur, un signal peut être envoyé aussi à un processus connu par son nom. La commande `killall` est équivalente à la fonction de la commande `kill` sauf qu'elle sélectionne tous les processus dont le nom correspond à celui donné en argument.

✍ **Challenge C24Q2**

Dur à la tâche

À noter que pour des raisons évidentes de sécurité, l'utilisation de la commande `kill` sur les processus d'un autre utilisateur est réservée au super-utilisateur du système.

Quand l'utilisateur n'a pas la possibilité de saisir une ligne de commande, par exemple lorsque son terminal est bloqué par une commande, il lui reste comme possibilité le contrôle du processus par l'envoi de signaux par une combinaison de touches. Ainsi la combinaison CTRL+c envoie le signal d'interruption SIGINT. La combinaison CTRL+z que nous avons vue dans la section précédente envoie le signal STOP. L'envoi de signaux par combinaison de touches est plus pratique que la commande `kill` et ne peut s'utiliser que sur des commandes d'avant plan (dans le terminal).

✍ **Challenge C24Q3**

Kill Bill

L'utilisateur peut attribuer une priorité à un processus. La priorité est attribuée sous forme d'incrément à l'aide de la commande `nice`. La priorité va de −20 (le plus prioritaire) à 19 (la moins prioritaire). Par exemple, pour lancer la commande `sleep` en arrière plan avec une faible priorité on tapera :

```
nice -n 15 sleep 10 &
```

Vous pouvez vérifier cette priorité par la commande `ps -l`.

Comme le shell est le parent de tous les processus de l'utilisateur, lorsque le shell se termine à la fermeture de la session (déconnexion), tous les processus d'arrière plan (qui sont aussi fils) sont terminés. Lorsque la session se termine, le signal SIGHUP est envoyé à tous les processus fils.
La commande `nohup` rend un processus insensible aux signaux INT et HUP. C'est un moyen de détacher un processus d'arrière plan du shell. Cette commande trouve son utilité quand un programme

avec un traitement long est lancé. L'utilisateur peut demander que le traitement continue une fois sa session terminée. Par exemple soit la commande utilisateur `monpgm`, le lancement du programme en mode détaché s'effectue de la manière suivante :

```
$ nohup monpgm &
```

5 Conclusion

Vous venez de voir la notion de processus. Le Bash est un processus qui sert à lancer d'autres commandes. À chaque commande externe exécutée, un nouveau processus est créé. Ce qui démontre qu'un processus est une instance d'un programme en cours d'exécution. Nous avons également vu que le Bash offre la notion de multi-tâches à travers l'exécution en parallèle des commandes. Ces tâches sont gérées par des commandes spécifiques. En conclusion, vous savez maintenant contrôler les commandes en cours d'exécution et vous pouvez lancer en tâche de fond des commandes et continuer à interagir avec votre shell pendant ce temps-là.

Il n'y a pas à dire, vous commencez à maîtriser le shell Bash !

Solutions des exercices

Solution de l'exercice 2.4.1 page 119:
1/ Il s'agit de la commande `ps`. Nous voyons que la sortie contient le pourcentage de CPU. Ceci s'obtient en précisant en option les usages. La commande complète est `ps u`
2/ Le processus de PID 103 est passé dans l'état arrêté. le signal SIGSTOP (ou STOP en abrégé) a été envoyé par la commande `kill -STOP 103`
3/ Les tâches d'exécutions ont été lancées par la ligne de commande :

```
$ sleep 150 & sleep 250 &
$ jobs
[1]-  Running                 sleep 150 &
[2]+  Running                 sleep 250 &
$ kill -SIGSTOP %1
```

Le caractère `&`, en plus d'envoyer en arrière plan la commande, est un séparateur de commandes. L'affichage de la liste des tâches d'exécutions s'obtient par la commande `jobs`. L'arrêt de la tâche 1 est effectué en envoyant le signal SIGSTOP à cette tâche indiquée par `%1`.
4/ La tâche d'exécution numéro 2 est supprimée par la commande `kill %2`.
5/ Dans cette question, il s'agit de redémarrer une tâche arrêtée. La commande est `kill -CONT %1`. Le passage de la tâche en premier plan s'effectue par la commande `fg %%`.
Note : la dernière tâche indiquée peut se désigner par l'écriture `%%`. On parle de tâche courante. Elle apparaît dans la liste des tâches produite par la commande `jobs` avec le symbole `'+'` associé au numéro de la tâche.

Activité 2.5

Entrées et sorties des processus

1 Introduction

Rappelons ce que nous avons appris de l'activité précédente. Chaque commande saisie dans le shell s'exécute sous la forme d'un processus. Comme une ligne de commande ne se limite pas à une seule commande, les commandes peuvent s'enchaîner les unes à la suite des autres. Pour effectuer des enchaînements de commandes, il faut pouvoir contrôler l'entrée et la sortie des commandes. C'est l'objectif de cette activité. Nous commencerons par indiquer les différents moyens proposés par le shell pour effectuer des enchaînements de commandes. La notion de canal pour l'entrée et la sortie d'un processus sera ensuite posée. Puis nous verrons comment ces canaux peuvent être redirigés vers des dispositifs choisis par l'utilisateur. Enfin, nous étudierons comment brancher les commandes les unes à la suite des autres.

1.1 Enchaînements de commandes

Nous avons vu dans l'activité précédente que lorsqu'une commande se termine par le caractère `'&'` , le shell exécute la commande en arrière plan. La fin de la commande n'est pas attendue pour lancer la suivante. Ainsi `cmd1 & cmd2 &` est une ligne de commande qui lance deux commandes en arrière plan.

Maintenant, pour exécuter des commandes séquentiellement à partir de la même ligne de commande, cela consiste simplement à exécuter les commandes les unes à la suite des autres. S'il n'y a aucune relation entre les commandes, les commandes sont séparées par le caractère `';'`. Ainsi `cmd1 ; cmd2` est une ligne de commande qui lance la seconde commande quand la première est terminée. Si il y a une condition sur l'exécution pour l'enchaînement des commandes, les commandes peuvent s'enchaîner en fonction du code retour de la commande précédente. On parle ici des opérateurs `&&` et `||` placés entre les commandes. Nous approfondirons leur utilisation dans l'activité 4.3.

En plus de la dépendance sur l'exécution de commande, il peut y avoir aussi une dépendance sur les données. C'est-à-dire que les données produites par une commande servent de données d'entrée pour la suivante. Nous pouvons imaginer enchaîner des commandes en utilisant un fichier intermédiaire : la première commande écrit son résultat dans un fichier que la deuxième commande utilise comme entrée. La directive d'utilisation d'un fichier repose sur le **mécanisme de redirection**. Après cette activité, l'utilisation de la redirection n'aura plus de secret pour vous.

Enfin, au lieu de passer par un fichier intermédiaire pour enchaîner les commandes, nous pouvons les brancher directement entre elles. Mieux que d'exécuter les commandes les unes après les autres, ce sont les données qui vont être traitées, par le branchement de commandes, au fur et à mesure de la lecture des données d'entrée. On appelle cela un **tube**, nous allons découvrir son utilisation.

1.2 L'entrée et les sorties d'une commande

Un processus, autrement dit une commande en cours d'exécution, utilise des canaux pour lire et écrire des données. Le canal pour la lecture des données passe par l'entrée standard. Sur le même principe, l'écriture des résultats de la commande passe par la sortie standard. Par défaut, l'entrée standard est associée au clavier et la sortie standard à l'écran du terminal. De façon plus discrète, une commande possède un troisième canal appelé sortie d'erreur qui est, par défaut, lui aussi associé à l'écran du terminal. Ce canal reçoit les éventuels messages d'erreurs de la commande. Vous trouverez très souvent une autre appellation pour ces canaux, il s'agit de *stdin* (*standard input*), *stdout* (*standard output*) et *stderr* (*standard error*).

De manière plus générale, ces trois canaux peuvent être associés à des fichiers de texte ; on parle alors de redirection des entrées-sorties. La figure 2.5.1 récapitule les entrées-sorties standards avec leurs associations par défaut ainsi que leur numéro associé. Ces numéros identifient les canaux. Ils sont utilisés dans les redirections détaillées par la suite.

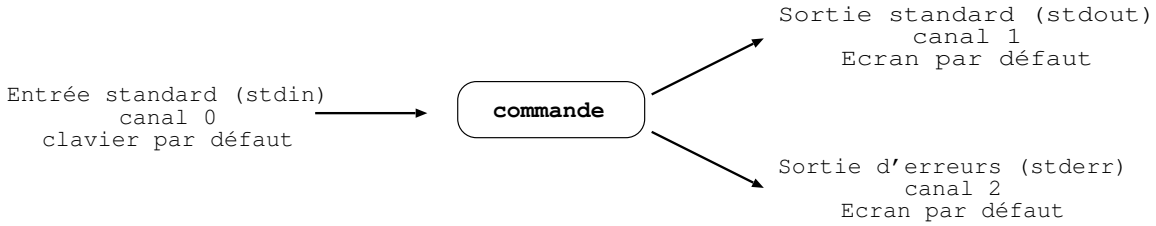


FIGURE 2.5.1 – Entrée et sorties standards.

2 Les redirections

Gérées par le shell, les redirections des entrées-sorties permettent de changer l'association par défaut des canaux de la commande. Il s'agit d'utiliser des fichiers de texte en entrée et en sortie plutôt que le clavier et l'écran. Ainsi, avec une redirection en sortie, le résultat d'une commande est conservé dans un fichier.

2.1 Sortie standard

Il existe deux modes de redirection selon qu'on veut envoyer la sortie d'une commande vers un nouveau fichier ou bien ajouter la sortie à la fin d'un fichier. Ces redirections sont indiquées par les opérateurs suivants :

`>` *fichier* écrit la sortie de la commande dans *fichier*. Si *fichier* existe, il est écrasé.
`>>` *fichier* ajoute la sortie de la commande à la fin de *fichier*. Si *fichier* n'existe pas, il est créé.

À titre d'exemple, étant connecté avec le compte d'Alice et situé dans le répertoire `/home/alice/Sequence2`, on peut rediriger la sortie de la commande `ls` dans un fichier de la manière suivante :

```
$ ls > liste.txt
```

Nous pouvons vérifier que le fichier a été créé et que son contenu correspond bien au résultat de la commande `ls` :

```
$ cat liste.txt
A21
A22
A23
```

```
A24
A25
liste.txt
```

Le lecteur attentif remarquera que le fichier `liste.txt`, issu de la redirection en écriture de la commande `ls`, a été listé. Cela veut dire que le fichier a été créé avant que la commande `ls` soit exécutée.

Une redirection en sortie peut aussi se faire en ajoutant le résultat à la fin d'un fichier existant. Ceci se spécifie avec l'opérateur de redirection `'>>'`. La commande ci-dessous ajoute le résultat de la commande `echo` à la fin du fichier `liste.txt`.

```
$ echo "-----------" >> liste.txt
```

Observons le résultat, et nous voyons bien qu'à la fin du fichier des tirets ont été ajoutés.

```
$ cat liste.txt
A21
A22
A23
A24
A25
liste.txt
-----------
```

2.2 Sortie d'erreur

Une commande possède une deuxième sortie qui reçoit les messages d'erreur écrits par la commande. Ce canal de sortie porte le numéro 2. La redirection de ce canal se note `2>`. Attention, soyez rigoureux dans l'usage des espaces, il n'y a pas d'espace entre le 2 et le `>`. Prenons un exemple, toujours à partir du répertoire `/home/alice/Compagnon`. Essayons de lister un fichier qui n'existe pas :

```
$ ls fich1
ls: fich1: No such file or directory
```

On voit s'afficher le message d'erreur sur l'écran, qui est, rappelons-le, le fichier associé par défaut à la sortie d'erreur.

Vous pouvez préférer sauver les messages d'erreurs dans un fichier pour en faire une analyse plus tard ; pour cela, il suffit de rediriger la sortie d'erreur dans un fichier :

```
$ls fich1 2>log.txt
```

Cette fois, rien ne s'affiche à l'écran et en consultant le contenu du fichier `log.txt`, nous pouvons vérifier que le message d'erreur a été récupéré.

```
$ less log.txt
ls: fich1: No such file or directory
```

La redirection de la sortie d'erreur peut aussi se faire en mode ajout en utilisant `2>>` :

```
$ cat fich1 2>>log.txt
```

Le fichier `fich1` n'existe toujours pas et on récupère le message d'erreur à la fin du fichier `log.txt` :

```
$ less log.txt
ls: fich1: No such file or directory
cat: can't open 'fich1': No such file or directory
```

Il arrive parfois que l'on souhaite ignorer l'affichage des erreurs, c'est-à-dire qu'on ne veut pas que les éventuelles erreurs s'affichent à l'écran, et on ne veut pas non plus les récupérer dans un fichier. Dans ce cas, il suffit de rediriger la sortie d'erreur vers le « trou noir » `/dev/null` qui est un fichier un peu particulier dans lequel tout ce qui est écrit est ignoré. Aucun message d'erreur ne s'affiche, c'est ce que nous voulions.

```
$ cat fich1 2>/dev/null
```

Il y a aussi la possibilité de rediriger un canal vers un autre. Par exemple, nous voulons que la sortie standard s'affiche dans le canal de la sortie en erreur. Cette opération s'indique de la manière suivante `1>&2` (sans espace entre `>` et le `&`). Elle se comprend comme le canal de numéro 1 est redirigé vers le canal de numéro 2 (représenté par le symbole `&2`). Cette redirection s'écrit encore plus simplement `>&2`). L'absence de numéro à gauche de l'opérateur de redirection équivaut à la sortie standard (numéro 1). Cette redirection trouve son utilité pour afficher des messages dans la sortie d'erreur comme :

```
$ echo "ERROR: Integer is expected" >&2
```

Pour regrouper les sorties standard et erreur d'une commande dans le même fichier, il faut rediriger les canaux par cette opération :

```
 2>fichier 1>&2
```

Dans cette opération, il faut que le nom du fichier en écriture n'apparaisse qu'une seule fois. De plus, il faut commencer par indiquer la redirection d'un canal vers le fichier puis rediriger l'autre canal vers le canal redirigé préalablement. Comme l'ordre des redirections est important, le shell propose une écriture simplifiée pour cette opération `&>`*fichier*. Par exemple, la commande `ls` ci-dessous exécutée dans le répertoire `/home/alice/Compagnon` va mettre dans le fichier `result.txt` un message d'erreur pour le répertoire inconnu et le contenu du répertoire `A33`.

```
$ ls dir A33 &>result.txt
$ cat result.txt
ls: dir: No such file or directory
A33:
Fichier1.txt fichier2.txt
```

Terminons la redirection de sortie pour signaler que l'écrasement du fichier existant est une source de belles boulettes ! Supposons qu'Alice édite un fichier pour y stocker des données importantes. Elle nomme ce fichier `result.txt`. Puis un peu plus tard, elle exécute une commande avec une redirection de la sortie dans un fichier qu'elle appelle par inadvertance `result.txt`. La redirection indiquée par le shell va écraser le fichier de données d'Alice sans aucune forme d'avertissement. Pour changer ce comportement, il faut positionner l'option `noclobber` (pas d'écrasement) du shell par la commande `set -o noclobber` !. Dans ce cas si le fichier existe, la redirection en sortie mettra la ligne de commande en erreur. Pour écraser le fichier il faut alors le demander explicitement par l'opérateur `>|`. Voyons cela avec l'exemple ci-dessous :

```
$ set -o noclobber
$ ls >result.txt
$ ls >result.txt
-bash: result.txt: cannot overwrite existing file
$ ls >|result.txt
```

```
$ set +o noclobber
```

La première commande supprime le mode écrasement du shell pour les redirections en sortie. La première redirection crée le fichier `result.txt`. La seconde redirection conduit à une ligne de commande en erreur car le fichier existe déjà et l'écrasement n'est plus possible. La troisième redirection demande explicitement d'écraser le fichier si celui-ci existe. La dernière commande remet le shell avec le mode écrasement.

2.3 L'entrée standard

Nous avons vu la redirection des sorties, maintenant regardons pour l'entrée. Cette fonctionnalité trouvera sa principale utilité dans des scripts pour automatiser des travaux comme vous le verrez dans la dernière séquence.

La redirection de l'entrée standard s'effectue avec le symbole `<`. Pour illustrer ce mécanisme, nous allons utiliser la commande `wc` (*Word Count*) qui effectue des comptages sur le contenu d'un fichier. En l'absence d'argument, le comptage s'effectue sur l'entrée standard, c'est-à-dire par défaut depuis le clavier ; voyons cela :

```
$ wc -l
un
deux
trois
```

CTRL + d

```
3
```

La combinaison de touches CTRL + d indique la fin de fichier, en l'occurrence la fin de la saisie au clavier. Juste après, on obtient le résultat de la commande `wc -l` qui a compté trois lignes.

À la place du clavier, nous pouvons alimenter l'entrée par un fichier en redirigeant l'entrée standard de la manière suivante :

```
$ wc -l <log.txt
2
```

La commande affiche le nombre de lignes contenues dans le fichier `log.txt`.

Remarque : le résultat est équivalent à celui obtenu sans redirection mais en donnant le nom du fichier à traiter en argument de la commande :

```
$ wc -l log.txt
2 log.txt
```

À l'exception cette fois-ci que le nom du fichier apparaît dans le résultat affiché. La commande `wc` affiche en effet dans son résultat le nom du fichier traité, nom qui est connu lorsqu'il est donné en argument mais que la commande ne peut pas connaître lorsque les données proviennent d'une redirection.

Il est aussi possible de rediriger l'entrée standard avec les doubles chevrons `'<<'`. Il ne s'agit pas là d'un mode « ajout en entrée » qui n'aurait pas beaucoup de sens mais d'un mécanisme qui permet d'indiquer à une commande que tout ce qui suit `<<` doit être pris comme entrée de la commande jusqu'à la rencontre d'une étiquette. Cette étiquette marquant la fin de l'entrée est indiquée juste après `<<`. Ainsi, dans l'exemple suivant, l'entrée de la commande `wc` est constituée de tous les caractères tapés entre les deux occurrences du mot `FIN` :

```
$ wc -l <<FIN
> un
> deux
> trois
> quatre
> FIN
4
```

Les chevrons qui apparaissent dans l'exemple ci-dessus devant chaque ligne ne sont pas à taper, ils sont affichés par le Bash qui signale ainsi que la ligne de commande n'est pas terminée. L'entrée ne provient donc pas du clavier comme lorsqu'on utilise `wc` sans argument ni redirection ; les données en entrée sont lues directement sur la ligne de commande.

Attention, l'étiquette utilisée comme marqueur du début et de fin de texte ne doit être ni précédée ni suivie d'autres caractères. Le premier mot suivant `<<` est pris comme étiquette de début. Cette étiquette doit être écrite seule sur une ligne pour marquer la fin de l'entrée. Cette dernière ligne ne fait pas partie de l'entrée de la commande.

En shell interactif, la redirection `<<` n'a pas beaucoup d'intérêt. Tout du moins, est-elle équivalente à une lecture depuis l'entrée standard avec indication de fin de fichier avec CTRL + d. La double redirection trouve son utilité dans un script pour rediriger l'entrée standard d'une commande d'un script à partir du script lui-même. Vous verrez ceci plus en détail à la fin de ce cours.

Exercice 2.5.1: Redirection en tous sens
Alice fait des redirections dans ses commandes. Pour chaque commande de la liste ci-dessous, saurez-vous indiquer le contenu du fichier `result.txt` ?

1. `cat lipsum.txt lipsum.txt >result.txt`
2. `echo "---" >>result.txt`
3. `cat inconnu >result.txt 2>&1`
4.
```
cat >result.txt  <<EOF
bonjour
EOF
```
5. `wc -w <lipsum.txt >result.txt`

Solution page 131

3 Branchement de commandes

Les commandes peuvent se brancher à la suite les unes des autres pour effectuer des traitements spécifiques. Le principe consiste à ce que la sortie d'une première commande soit transférée à l'entrée de la commande suivante, sans passer par un fichier intermédiaire. Un tube (*pipe* en anglais) est alors établi entre les 2 commandes. L'opérateur du tube est la barre verticale (`|`). À titre d'exemple, nous pouvons compter le nombre de commandes dans l'historique :

```
$ history | wc -l
5449
```

En branchant la sortie standard de `history` sur l'entrée de la commande `wc`, le résultat final est un comptage.

Pour réaliser la même opération sans tube, nous pourrions écrire deux commandes avec deux redirections. On redirige d'abord la sortie de `history` vers un fichier qu'on utilise ensuite comme entrée de la commande `wc`. Cela nous oblige cependant à créer un ficher inutile :

```
$ history >fichier; wc -l <fichier
5449
```

Si le résultat de l'opération est identique, sa mise en œuvre est différente. Avec un tube, les commandes branchées créent des processus existants simultanément. Les processus s'exécutent de manière synchronisées et non les uns après les autres. Les données sont traitées en flux. C'est à dire quand une donnée est produite sur la sortie d'une commande, elle alimente l'entrée de la commande suivante branchée sur sa sortie. Ainsi toutes les commandes branchées s'exécutent en même temps en traitant le flux de données au fur et à mesure qu'il est produit.

On peut voir la commande `cat` comme une commande qui concatène des flux. Prenons par exemple, l'enchaînement de commandes par tube suivant :

```
$ ps -f | cat fichier1.txt - | less
```

Ici la commande `cat` prend deux flux en entrée : un provenant d'un fichier et l'autre provenant du tube qui est indiqué par le caractère tiret (`'-'`). Ce dernier est généré par la commande `ps`. La sortie de la commande `cat` est passée au pageur `less`. Si vous regardez la dernière page affichée. Vous lisez ceci :

```
UID        PID  PPID  C STIME TTY          TIME CMD
alice       75     1  0  1903 ttyS0    15049-12:58:15 -bash
alice       87    75  0  1903 ttyS0    15049-12:58:15 ps -f
alice       88    75  0  1903 ttyS0    15049-12:58:15 cat fichier1.txt -
alice       89    75  0  1903 ttyS0    15049-12:58:15 less
```

Vous pouvez remarquer que les trois processus des commandes branchées existent en même temps. Ceci montre que c'est bien un flux de données qui passe par les tubes et non des fichiers temporaires.

Le branchement de commandes n'a de sens que pour les commandes capables de lire et d'écrire depuis les canaux standards. Cette possibilité de brancher les commandes est une des fonctionnalités les plus importantes du Bash, elle permet de construire des commandes sophistiquées en branchant plusieurs commandes élémentaires. On n'est bien sûr pas limité à seulement deux commandes, on peut brancher la sortie d'une première commande sur l'entrée d'une deuxième et brancher la sortie de cette deuxième commande sur l'entrée d'une troisième, etc.

Vous aurez l'occasion de mettre en œuvre cette fonctionnalité du Bash dans la séquence suivante de ce cours.

Exercice 2.5.2: En une ligne
Alice commence à percevoir l'intérêt du branchement. Elle décide d'écrire des lignes de commande avec des branchements. Expliquer le résultat de chacune des lignes ci-dessous :

1. `echo "---" | cat lipsum.txt - >result.txt`
2. `man head | head -n 5 | tail -n 1`
3. `date | cat lipsum.txt - >result.txt ; cat lipsum.txt | wc -l >>result.txt`
4. `cat inconnu  2>&1 | wc -c`

Solution page 131

✍ **Challenge C25Q1**

redirection et branchement

✍ **Challenge C25Q2**

Enregistrement en bonne forme

4 Conclusion

Cette activité vous a permis de voir comment :

- enchaîner séquentiellement des commandes en les séparant par un point-virgule
- rediriger l'entrée et les sorties standards d'une commande :
  - `>` redirige la sortie standard vers un fichier qui est créé ou écrasé ;
  - `>>` ajoute la sortie standard dans un fichier existant (et le créer s'il n'existe pas) ;
  - `<` permet de lire l'entrée depuis un fichier ;
  - `<<` permet de prendre tous les caractères entre deux mots marqueurs comme données d'entrée d'une commande ;
  - `2>` et `2>>` sont utilisés pour rediriger la sortie d'erreur ;
- et surtout à brancher des commandes entre elles à l'aide d'un tube `|`.

Solutions des exercices

Solution de l'exercice 2.5.1 page 128:
1/ Le fichier `result.txt` comporte le fichier `lipsum.txt` en double.
2/ La ligne de tirets est ajoutée en fin du contenu actuel du fichier `result.txt`.
3/ On suppose que le fichier `inconnu` n'existe pas. La commande `cat` rencontre une situation d'erreur. Le message d'erreur est envoyé sur la sortie d'erreur (identifiée par le numéro 2). La sortie standard est redirigée vers le fichier `result.txt` puis la sortie d'erreur est redirigée vers la sortie standard (qui a été préalablement redirigée dans le fichier). Au final, les deux sorties de la commande sont redirigées dans le fichier `result.txt`. Par conséquent, le fichier `result.txt` comporte uniquement le message d'erreur de la commande `cat`. Il n'y a aucun contenu obtenu d'un fichier qui n'existe pas ! Le Bash propose une écriture plus simple pour rediriger les deux canaux de sorties vers le même fichier de cette manière : `cat inconnu &>result.txt`
4/ En l'absence d'argument avec la commande `cat`, la commande lit les données de l'entrée standard (le clavier). La sortie est redirigée vers le fichier `result.txt`. Lorsqu'il y a une double redirection en lecture, l'entrée de la commande est prise sur ce suit l'étiquette (qui est ici EOF). L'entrée se termine dès que cette étiquette est seule en début de ligne. Ainsi le fichier `result.txt` contient la ligne « bonjour » uniquement.
5/ Le fichier `result.txt` contient le nombre de mots du fichier `lipsum.txt`. Comme la commande `wc` effectue le comptage depuis l'entrée standard qui est redirigée en lecture sur le fichier `lipsum.txt`, le fichier `result.txt` contient uniquement un nombre de mots (sans nom de fichier associé).

Solution de l'exercice 2.5.2 page 129:
1/ La sortie de la commande `echo` est branchée sur l'entrée standard de la commande `cat`. Cette commande concatène le contenu du fichier `lipsum.txt` avec le flux de données sur l'entrée standard. La position du flux de données de l'entrée standard dans la liste des fichiers à concaténer (les arguments de `cat`) est indiquée par le caractère tiret (`'-'`). Le résultat de cette concaténation est mis dans le fichier `result.txt`.
Le même résultat aurait été obtenu avec ces 2 commandes :

```
cat lipsum.txt >result.txt
echo "---" >>result.txt
```

2/ Le manuel de la commande `head` est passé comme entrée de la commande `head` qui ne garde que les 5 premières lignes. Ces 5 lignes sont ensuite passées comme entrée à la commande `tail` qui n'affiche que la dernière ligne. Au final, c'est la cinquième ligne du manuel de `head` qui est affichée.
3/ Cette ligne de commande comporte en fait deux parties qui s'enchaînent séquentiellement. Les 2 parties sont séparées par un point virgule. En premier, la date est concaténée avec le fichier `lipsum.txt` pour former le fichier `result.txt`. Une fois la première partie terminée, la deuxième partie de la ligne de commande est lancée. Cette partie comporte un branchement de commande pour effectuer le comptage du nombre de lignes du fichier `lipsum.txt`. Le résultat est ajouté en fin de fichier de `result.txt`. Une écriture équivalente de la deuxième partie est :

```
wc -l <lipsum.txt >>result.txt
```

4/ Le fichier inconnu n'existe pas. L'affichage de son contenu produit une erreur. Le canal de sortie d'erreur de la commande `cat` est redirigé sur le canal de sortie standard. Et ce canal est ensuite branché sur l'entrée de la commande `wc` pour le comptage des octets. Cette commande affiche finalement le nombre d'octets du message d'erreur de la commande `cat`.

Conclusion

À l'issue de cette séquence, vous voilà pleinement opérationnel pour interagir avec le shell Bash. Vous pouvez maintenant :

- éditer ou rappeler une ligne de commande,
- désigner un groupe de fichiers avec des noms abrégés,
- indiquer des arguments à une commande sous la forme de variables ou de résultat d'une autre commande,
- contrôler l'exécution d'une commande,
- gérer les canaux d'entrée et de sortie d'une commande. Ainsi une commande peut alimenter soit un fichier, soit une autre commande directement.

Grâce à ces nouvelles notions, vous pouvez maintenant utiliser le Bash comme une interface utilisateur. Vous pouvez maintenant examiner une arborescence de fichiers mais également contrôler l'exécution d'un processus. Vous pouvez aussi enchaîner des commandes pour constituer des lignes de commandes qui répondent à des actions plus spécifiques à effectuer. Bref vous connaissez les structures du langage shell Bash pour son utilisation interactive.

Il est temps maintenant d'étendre vos compétences avec des nouvelles commandes capables de traiter et manipuler des données. Vous allez découvrir et apprendre à utiliser la boite à outils du shell. Cela va être l'objectif de la prochaine séquence.

Continuez, vous êtes sur la bonne voie pour exploiter toute la puissance et la flexibilité du Bash.

Séquence 3

Maîtrisez votre système d'exploitation grâce au Bash

Introduction

Vous avez découvert votre système d'exploitation au travers de la ligne de commande et vous savez maintenant interagir avec votre système d'exploitation. Vous savez comment fonctionne votre système d'exploitation et savez exploiter quelques-unes des ses fonctionnalités en utilisant la ligne de commande.

Dans cette séquence, nous allons apprendre à utiliser des outils élémentaires fournis avec votre système. Avec ces outils, vous allez pouvoir traiter le flux de données produit par certaines commandes et ainsi devenir plus efficace et performant en filtrant les informations non pertinentes. Ces outils ne vont plus vous quitter et vont rendre la ligne de commande indispensable pour vos actions courantes.

Dans un premier temps, nous allons découvrir l'environnement dans lequel nous travaillons et nous allons apprendre à l'adapter à nos habitudes de travail. Nous allons ensuite apprendre à manipuler des fichiers car tout est fichier sous Unix. Nous allons d'abord essayer d'en extraire simplement des informations pertinentes puis nous allons utiliser des outils pour effectuer des tâches complexes et puissantes sur ces fichiers. Ensuite, nous allons découvrir comment effectuer des calculs avec le terminal et enfin comment préparer vos fichiers pour l'archivage et/ou le transfert.

L'objectif de cette séquence est de faire en sorte que vous ne quittiez plus jamais votre ligne de commande car les outils qui y sont introduits vous feront gagner en efficacité. L'objectif est aussi de vous faire passer du monde des débutants sous Unix au monde des "avancés".

Activité 3.1

Contrôler son environnement

1 Introduction

Vous êtes désormais confortable devant votre terminal et en face de votre shell. Vous voulez maintenant être efficace et performant. Le shell et l'invite de la ligne de commande ne vous permettent pas d'exploiter tout votre potentiel. Comme tout logiciel ou interface utilisateur, il faudrait que l'interface s'adapte à vous et non l'inverse. Utiliser et apprendre à utiliser le shell, comme beaucoup de logiciels, peut être long et complexe mais si vous pouviez modifier le logiciel pour qu'il s'adapte à vous, cela facilitera cet apprentissage. Contrairement à beaucoup de logiciels, le shell peut se configurer pour s'adapter à vos besoins et simplifier les actions courantes que vous effectuez.

Durant cette activité, vous allez découvrir l'environnement de votre shell et apprendre à lister et afficher les variables de votre environnement. Vous allez aussi découvrir les commandes de consultation de l'environnement. Vous allez apprendre à créer des variables et à les exporter (la notion d'exportation sera définie et utilisée) pour qu'elles fassent partie de votre environnement de travail. Vous allez apprendre à les modifier, et configurer finement votre environnement. Vous pourrez ainsi adapter votre environnement à vos habitudes et exploiter efficacement votre shell.

2 Variables et options de l'environnement de travail

Pour rappel, en programmation, on peut créer des variables qui sont l'association de noms et de valeurs. Ces associations correspondent à des emplacements dans la mémoire de l'ordinateur. Associer nom et valeur donne la possibilité de traiter des données de manière symbolique. Dans les systèmes de type Unix, l'utilisateur peut créer et utiliser des variables pour profiter de cette association symbolique. Cependant, il faut distinguer les variables créées par l'utilisateur et celles créées et gérées par le shell lui-même. Les variables créées et gérées par le shell sont les variables d'environnement. Ces variables ont des noms prédéfinis et sont utilisées pour et par le shell. Elles sont renseignées par le shell lorsque l'utilisateur se connecte au système. Ces variables peuvent éventuellement être utilisées par d'autres programmes ou par l'utilisateur.

2.1 Variables d'environnement

Les variables d'environnement sont relatives à chaque utilisateur et à chaque session. C'est-à-dire qu'elles ne peuvent être utilisées et exploitées que dans la session courante. Bien que ces variables soient automatiquement renseignées par le shell (ou initialisées si on utilise un langage de programmation), leur valeur est modifiable par l'utilisateur. Ces modifications influent sur le comportement du shell

lors de la session courante. Il existe de nombreuses variables, nous n'allons pas ici en faire le catalogue. Cependant nous citons ci-dessous quelques noms de variables à retenir.

```
HOME     : Répertoire personnel de l'utilisateur
PATH     : Les répertoires contenant les commandes utilisables
LOGNAME  : Nom de l'utilisateur à la session
SHELL    : Type de shell utilisé à l'ouverture de la session
```

La variable `PATH` est une variable importante, elle donne l'accès aux commandes du système. Cette variable sert à retrouver les commandes dans l'arborescence sans qu'il soit nécessaire de fournir le chemin d'accès absolu. La variable `PATH` représente une règle de recherche pour le shell. C'est la liste des répertoires dans lesquels le shell doit rechercher une commande (en suivant l'ordre des répertoires listés dans la variable).

Pour illustrer tout l'intérêt de cette variable prenons un exemple. Supposons que vous voulez afficher le contenu du répertoire courant. La commande `ls` pour réaliser votre action se trouve dans `/usr/bin`. Si la variable `PATH` est vide, pour utiliser la commande `ls`, vous devez saisir :

```
$ /usr/bin/ls
```

En revanche, si votre variable `PATH` contient le répertoire `/usr/bin/` alors vous pouvez saisir :

```
$ ls
```

Maintenant supposons que vous ayez deux commandes `ls`, l'une que vous avez créée et l'autre la commande classique. Celle que vous avez créée se trouve dans le répertoire `/usr/local/bin` (la commande classique se trouve toujours dans `/usr/bin`).
Si votre variable `PATH` contient `/usr/local/bin:/usr/bin/` alors en saisissant la commande `ls`, la version utilisée sera celle que vous avez créée car l'ordre compte dans la liste des répertoires de la variable `PATH`. Consulter le contenu d'une variable d'environnement s'effectue par une commande d'affichage `echo` et une substitution de variable comme le montre l'exemple avec la variable `PATH`.

```
$ echo $PATH
/usr/local/bin:/sbin:/bin:/usr/bin
```

La variable `HOME` contient le chemin d'accès du répertoire personnel de l'utilisateur. La commande `cd` sans argument utilise la variable d'environnement `HOME`. Avec cette commande, le répertoire courant va devenir le répertoire personnel de l'utilisateur. L'exemple suivant montre l'action de cette commande dans le cas `alice`.

```
$ pwd
/home/alice/Sequence3
$ echo $HOME
/home/alice
$ cd
$ pwd
/home/alice
```

Il est possible de modifier le contenu d'une variable d'environnement pour changer le comportement d'une commande qui utiliserait cette variable. Nous pouvons par exemple modifier le comportement de la commande `man`. La commande `man` utilise la variable d'environnement `MANPAGER` pour l'affichage de la page de manuel. Le contenu de la variable `MANPAGER` est donné par la commande suivante :

```
$ echo $MANPAGER
less -R
```

```
$ man ls
[...]
Affichage du manuel avec la commande less -R
[...]
```

Il est possible de modifier le comportement de la commande `man` pour qu'elle utilise la commande `cat` pour afficher la page de manuel. La suite de commandes suivante montre cette modification.

```
$ MANPAGER=cat
$ echo $MANPAGER
cat
$ man ls
[...]
Affichage du manuel avec la commande cat
[...]
```

Attention, la modification de la variable `MANPAGER` sera effective pour toute la session.

✍ **Challenge C31Q1**

Même commande, autre résultat

Exercice 3.1.1: Pour aller plus loin.

1. Que contient la variable `PWD` ?
2. Que contient la variable `OLDPWD` ?
3. Quelle commande utilise la variable `OLDPWD` ?

Solution page 149

2.2 Les options

Le comportement du Bash se modifie et se configure au moyens des options du shell. La commande `shopt` (*Shell option*) positionne les options. Sans argument, cette commande liste les options disponibles du shell et montre si ces options sont positionnées ou non.

```
$ shopt
[...]
cmdhist          on
extglob          off
login_shell      on
[...]
```

La commande précédente montre que l'option `cmdhist` (pour la sauvegarde des commandes multilignes dans l'historique) est activée alors que l'option `extglob` (voir activité 2.5) ne l'est pas.

Pour activer l'option `extglob`, il faut préciser l'option -s (*set*) de la commande `shopt` :

```
$ shopt -s extglob
```

A l'inverse, pour rendre une option inactive, il faut utiliser l'option -u (*unset*) de la commande `shopt` :

```
$ shopt -u extglob
```

Nous ne rentrerons pas plus en détail dans les options du shell. La commande `help -m shopt` donne une aide sur la commande `shopt` et les options disponibles.

3 Votre environnement de travail

L'utilisateur doit connaître l'environnement dans lequel il travaille pour éviter des erreurs comme pour l'utilisation de la commande `ls` vue dans la section précédente. Il doit le connaître aussi pour pouvoir le modifier et l'adapter à ses besoins et ses habitudes. Nous savons que dans le cas du shell, l'environnement de travail est défini par des variables actives ou définies dans la session de travail courante.

Deux commandes sont disponibles pour manipuler et voir cet environnement : `set` et `env`.

3.1 La commande env

Employée sans argument, la commande `env` présente les mêmes résultats que la commande `printenv`. La différence entre ces commandes est historique entre Unix et Linux. La commande `env` liste les variables de l'environnement courant. Utilisée avec des arguments, elle sert à modifier les valeurs des variables de cet environnement. De nombreuses variables ont été enlevées de l'exemple suivant.

```
$ env
PWD=/home/mylogin
SHELL=/bin/bash
PATH=/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
LC_CTYPE=UTF-8
TERM=xterm-256color
HOME=/home/mylogin
USER=mylogin
EDITOR=/usr/bin/vim
MANPAGER=less -R
```

Dans l'exemple précédent, la variable `MANPAGER` donne la commande pour afficher les pages de manuel. Dans l'environnement défini précédemment, la commande `man` utilise la commande `less -R` pour afficher la page de manuel.

```
$ man date
[...]
Affichage du manuel avec less -R
[...]
```

La commande `env` peut aussi s'utiliser avec des arguments.

```
$ env variable1=valeur1 variable2=valeur2 cmd opt args
```

La commande précédente exécute la commande `cmd` avec les options `opt` et les arguments `args` en modifiant ou en étendant l'environnement de travail actuel avec les variables `variable1`, `variable2` et leurs valeurs respectives. Cette extension/modification n'est valide que pour l'exécution de la commande `cmd`.

```
$ echo $MANPAGER
less -R
$ env MANPAGER=cat man date
Affichage du manuel avec cat
```

```
$ echo $MANPAGER
less -R
```

La première commande affiche le contenu de la variable `MANPAGER`. La seconde commande exécute la commande `man date` en modifiant la variable `MANPAGER`, pour utiliser la commande `cat`. La dernière commande montre que la valeur de la variable d'environnement n'a pas été modifiée.

Attention, dans certains systèmes (comme la Weblinux) il est possible d'omettre la commande `env` pour avoir le même comportement. Ainsi la commande ci-dessous fonctionne de la même manière.

```
$ MANPAGER=cat man date
Affichage du manuel avec cat
```

✍ **Challenge C31Q2**

Raccourci

3.2 Les commandes `set`, `unset` et `declare`

La commande `set` utilisée sans argument fournit les variables de l'environnement mais aussi les variables créées par l'utilisateur dans la session courante. Dans l'exemple suivant, de nombreuses lignes ont été supprimées.

```
$ set
[...]
PWD=/home/mylogin
PATH=/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
TZ=GMT
[...]
$ var=1
$ set
[...]
PWD=/home/mylogin
PATH=/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
TZ=GMT
var=1
[...]
```

Dans l'exemple précédent, la commande `var=1` crée une variable de nom `var` dont la valeur est `1`. Lors de la deuxième exécution de la commande `set`, cette variable et sa valeur sont listées et font donc partie de l'environnement.

La commande `unset` supprime une variable de l'environnement. La syntaxe de la commande `unset` est la suivante :

```
$ var=1
$ echo $var
1
$ unset var
$ echo $var

$
```

Remarque : Différences entre variables utilisateurs et variables d'environnement. Sous Unix il existe deux catégories de variable : Les variables utilisateurs et les variables d'environnement. Les variables utilisateurs ne sont valables que dans l'instance actuelle du shell. Elles sont utilisées pour modifier à court terme le comportement du shell. Les variables d'environnement sont valides pour toutes les sessions du shell. Par convention le nom des variables d'environnement est en lettres capitales.

La commande `declare` sans argument affiche comme la commande `set` toutes les variables de la session courante. La seule différence est que la commande `declare` différencie les variables exportées (voir sous-section suivante) et les variables locales.

La commande `set` comme la commande `shopt` permet aussi de modifier les options du shell. Nous ne rentrerons pas dans les détails de ces options et nous ne ferons pas un détail des différences entre ces deux commandes ici. La commande `set -o` permet de lister les options contrôlées par la commande `set`. Pour obtenir de l'aide sur cette commande : `help -m set`.

3.3 La commande `export`

Une variable créée par l'utilisateur dans un shell est dite locale. Elle n'est connue qu'à l'intérieur du shell qui l'a créée. Il peut arriver que l'on souhaite lancer un autre shell depuis le shell actuel. On appelle ce second shell un sous-shell. Les variables d'environnement du shell original sont transmises au sous-shell. Ainsi, si dans le shell d'origine la variable d'environnement `HOME` existe et est définie, dans le sous-shell cette variable existera aussi et aura la même valeur.

```
$ printenv
[...]
HOME=/home/mylogin
[...]
$ bash
$ printenv
[...]
HOME=/home/mylogin
[...]
```

Dans l'exemple ci-dessus, la deuxième commande `bash` lance le programme Bash dans le shell original. On vérifie que dans ce sous-shell, la variable d'environnement `HOME` est bien définie et a la même valeur que le shell original.

Supposons maintenant que l'on crée une variable `mytimezone` dans le shell original. Pour la rendre accessible au sous-shell, il faut ajouter la variable `mytimezone` aux variables d'environnement et ainsi la transformer (ou la promouvoir) en variable d'environnement. Cette transformation et cet ajout sont faits avec la commande `export` comme le montre l'exemple ci-dessous.

```
$ mytimezone=CET
$ echo $mytimezone
CET
$ bash
$ echo $mytimezone
```

Ci-dessus, la variable `mytimezone` n'est pas accessible au sous-shell.

```
$ mytimezone=CET
$ echo $mytimezone
CET
$ export mytimezone
$ printenv
```

```
[...]
mytimezone=CET
[...]
$ bash
$ echo $mytimezone
CET
```

4 Les fichiers de configuration du shell

Les systèmes de type Unix sont bien connus pour les possibilités de configuration qu'ils offrent. Les variables d'environnement peuvent être modifiées ou créées dès la connexion d'un utilisateur au système. Ces initialisations ou modifications peuvent être conservées dans des fichiers. Ainsi, si l'utilisateur veut qu'une variable d'environnement `mytimezone` soit créée à chacune de ses connexions, il pourra mettre la création et l'exportation de cette variable dans un fichier.

4.1 Les fichiers `/etc/profile` et `/.profile`

Quand vous vous identifiez, ou quand vous ouvrez une fenêtre terminal, des scripts shell spécifiques sont exécutés. Les scripts sont des fichiers contenant des commandes shell à exécuter.

Le fichier `/etc/profile` est un script exécuté automatiquement à la connexion pour tous les utilisateurs (la notion de script sera vue ultérieurement). Ce fichier définit les variables et paramètres universels identiques pour tous les utilisateurs. Attention, ce fichier est commun à plusieurs shells (sh, ksh, bsh, bash, $\cdots$). Il ne faut donc pas y mettre de variables spécifiques à Bash. Et si de telles variables spécifiques sont nécessaires il faut les mettre dans le fichier `/etc/bashrc`. L'exemple suivant montre un extrait de ce que peut contenir le fichier `/etc/profile`.

```
[...]
PATH="$PATH:/usr/local/bin:/usr/bin"
USER=‘id -un‘
LOGNAME=$USER
export PATH USER LOGNAME
[...]
```

Cet exemple montre que les variables d'environnement `PATH, USER, LOGNAME` sont définies et exportées en tant que variables d'environnement.

Le fichier `~/.profile` ou `/home/mylogin/.profile` contrairement au fichier `/etc/profile` n'est exécuté que lorsque l'utilisateur *mylogin* se connecte. Ce fichier est exécuté après l'exécution du fichier `/etc/profile` - il permet donc de modifier les variables déjà définies dans `/etc/profile` ou d'en créer de nouvelles. Il est modifiable par l'utilisateur et contient les variables d'environnement qui lui sont spécifiques. C'est dans ce fichier par exemple que doit être créée puis exportée notre variable `mytimezone`. Pour ce faire, on doit ajouter les lignes suivantes dans ce fichier :

```
[...]
mytimezone=CET
export mytimezone
[...]
```

4.2 Les fichiers /etc/bashrc et /.bashrc

Si le shell de l'utilisateur *mylogin* est Bash alors le script `/home/mylogin/.bashrc` est exécuté à la suite des fichiers `/etc/profile` et `/home/mylogin/.profile`. Le fichier `/home/mylogin/.bashrc` débute souvent par des commandes qui vont conduire à lancer l'exécution du fichier `/etc/bashrc`. Car ce dernier contient des variables, fonctions, ou alias propres à Bash et communs à tous les utilisateurs. La suite du fichier `/home/mylogin/.bashrc` est complétée par les définitions des variables, fonctions et alias spécifiques à l'utilisateur. Voici un extrait du contenu d'un fichier `/home/mylogin/.bashrc`

```
[...]
if [ -f "/etc/bashrc" ] ; then
  source /etc/bashrc
fi
export EDITOR=/usr/bin/vi
[...]
```

Les premières lignes testent si le fichier `/etc/bashrc` existe, et si c'est le cas l'exécutent avec la commande `source`. Puis, la variable `EDITOR` est définie et exportée. La valeur de cette variable est la commande de l'éditeur de texte par défaut de l'utilisateur.

✍ **Challenge C31Q3**

Raccourci permanent

Exercice 3.1.2: Pour aller plus loin.

1. Avec la variable `OLDPWD` comment copier le fichier `test.txt` du répertoire courant dans le répertoire visité précédemment ?
2. Quelle est, selon vous, l'utilité de la variable `PWD` alors que le résultat peut être obtenu avec la commande `pwd` ?

Solution page 149

5 L'invite de commande et le terminal

5.1 La variable PS1

Jusqu'ici, votre invite de commande est représentée par le caractère `'$'`. Les systèmes de type Unix permettent de modifier facilement cette invite de commande pour qu'elle contienne les informations qui vous sont utiles. Les informations pouvant être affichées dans l'invite de commande peuvent être le contenu de variables d'environnement ou le résultat produit par l'exécution de certaines commandes. La définition des informations affichées dans l'invite de commande est spécifiée dans la variable d'environnement `PS1`. Il faut noter que pour rendre les modifications de la variable `PS1` permanentes, celles-ci doivent être sauvegardées dans l'un des fichiers de configuration du shell, comme par exemple le fichier `/etc/profile`. Le code ci-dessous montre des modifications de la variable d'environnement `PS1` et leur effet sur l'invite de commande (le nom d'utilisateur est ici `alice`) :

```
$ PS1="\u > "
alice >
alice > PS1="\u dans \w > "
alice dans ~ > cd /tmp
alice dans /tmp >
```

Dans cet exemple, l'invite de commande `$` est d'abord remplacée par le nom d'utilisateur suivi d'une espace et le signe `>` suivi d'une espace. Puis, on modifie encore cette invite de commande pour qu'elle affiche le répertoire dans lequel se trouve l'utilisateur.

Les commandes pouvant être utilisées dans l'invite de commande pour les modifications de la variable `PS1` sont (liste non exhaustive) :

```
\u : login de l'utilisateur courant
\w : répertoire courant, avec $HOME abrégé par un tilde
\t : Heure courante au format 24-heures HH:MM:SS
\s : Le nom du shell, aussi retourné par la variable $0
\T : Heure courante au format in 12-heures HH:MM:SS
\@ : Heure courante au format in 12-heures AM/PM
\A : Heure courante au format in 24-heures HH:MM
\H : Nom de la machine
\W : Nom de base du répertoire courant, avec $HOME abrégé par un tilde
\! : Le numéro historique de la commande
\# : Le Numéro de la commande
\n : Passage à la ligne
```

✍ **Challenge C31Q4**

L'invite de commande

5.2 La commande `clear`

Maintenant que vous manipulez confortablement votre environnement et que vous avez saisi plusieurs commandes dans votre terminal, ce dernier est rempli de texte. La commande `clear` peut alors vous être utile, car elle permet de vider votre terminal. Un équivalent à cette commande est la combinaison de touches CTRL+L.

5.3 Les multiplexeurs de terminaux : `tmux`

Nous avons vu plus haut que l'on pouvait lancer la commande `bash` dans un terminal et que cela crée un sous-shell. Nous avons vu aussi dans l'activité 2.4 comment contrôler les processus. Imaginons un scénario dans lequel on souhaite lancer la commande `bash`. L'utilité d'exécuter ce sous-shell est de pouvoir modifier les variables d'environnement sans pour autant affecter les variables du shell d'origine. Une fois le sous-shell lancé, on lance une commande quelconque dans ce sous-shell qui prend du temps à s'exécuter. Dans notre scénario, nous ne voulons pas attendre la fin de cette commande pour revenir à notre shell d'origine. La solution, comme dans l'activité 2.4, c'est de mettre l'exécution du sous-shell en tâche de fond.

Pour cela, nous rappelons qu'il existe un outil très pratique mais qui n'est pas installé par défaut sur les systèmes de type Unix. C'est l'outil `tmux`[1], déjà présenté lors de l'activité 2.4. Cet outil permet, entre autres, de résoudre le problème décrit ci-dessus en permettant de mettre un shell tout entier en tâche de fond (en arrière-plan) ou de le ramener en avant-plan.

```
$ tmux
$ PATH=/usr/local/bin  # modification de variables d'environnement
$ ls -Ral /            # commande dont le traitement est très long
```

1. Cet outil n'est pas présent sur la Weblinux

```
<CTRL>+b d        (actions sur le clavier pour passer la session tmux en arrière
    plan (detach))
$ tmux a              # ramène la session en avant plan (attach)
```

6 Conclusion

Dans cette activité nous avons vu comment configurer et modifier notre shell et les variables d'environnement. Cette activité ne montre qu'un aperçu de ce qu'il est possible de faire pour contrôler son environnement. Au fur et à mesure de votre utilisation du shell, vous modifierez les variables d'environnement et exploiterez de plus en plus les fichiers de configuration de votre shell. Les possibilités de modification et d'adaptation des variables sont infinies. Plus vous utiliserez ces raffinements et plus vous serez rapide, efficace et performant dans l'ensemble des usages du shell.

Solutions des exercices

Solution de l'exercice 3.1.1 page 141:

1. La variable `PWD` contient le chemin complet du répertoire courant.
2. `OLDPWD` contient le répertoire précédemment utilisé. Par exemple, placez-vous dans le répertoire `/tmp` avec la commande `cd /tmp`, puis placez vous dans le répertoire `/home/alice/Documents` avec la commande `cd /home/alice/Documents`. Après ces deux commandes la variable `OLDPWD` contient normalement `/tmp` qui était votre répertoire précédent.
3. la commande `cd -` replace le répertoire courant dans le répertoire précédent.

Solution de l'exercice 3.1.2 page 146:

1. `cp test.txt $OLDPWD`
2. L'utilisation de variables est plus facile et performante dans l'écriture de script que l'exécution d'une commande. Vous verrez les scripts dans la séquence 4.

Activité 3.2

Filtres simples

1 Introduction

Vous savez lire le contenu d'un fichier grâce aux commandes `vi, less, cat`... Dans les systèmes de type Unix, tout est considéré comme fichier. Il y a plusieurs types de fichier mais ceux qui nous intéressent dans cette activité ce sont les fichiers contenant du texte ou à partir desquels on peut extraire du texte. Nous nous intéressons aussi aux sorties d'une commande qui produit du texte. Souvenez-vous que dans l'activité 2.5, nous avons vu que la sortie d'une commande peut être exploitée comme un fichier lu par une autre commande grâce à la syntaxe de branchement de commande (|).

Quand on traite des fichiers, toutes les informations ne sont pas toujours pertinentes à tout moment. Il est donc important de pouvoir isoler les informations pertinentes, ou d'être en mesure d'organiser le fichier d'une manière plus accessible, ou encore d'être capable d'obtenir des informations liées au contenu du fichier. Supposons que vous ayez un fichier de plusieurs centaines de milliers de lignes, en extraire une information pertinente peut s'avérer complexe. Des commandes du shell permettent de faire ce travail simplement et rapidement. Cette activité s'intéresse à ces commandes-là. Nous allons en effet voir ici comment extraire des informations des fichiers texte, et nous allons pour cela utiliser des filtres simples.

Un filtre est un moyen, comme son nom l'indique, de simplifier, séparer, épurer ou clarifier un flux de données. Un flux de données est souvent un texte ou une chaîne de caractères. Les commandes qui lisent et écrivent sur les E/S sont appelées des filtres. Les filtres sont souvent utilisés en branchement de commande (en réalisant un tube avec l'aide du caractère |).

2 Découpage d'un fichier

2.1 Les commandes `head` et `tail`

Comme nous l'avons vu dans l'activité 1.5, les commandes `head` et `tail` permettent de n'afficher que les premières ou dernières lignes d'un fichier ou de l'entrée standard. Les syntaxes des deux commandes sont les suivantes :

```
$ head [options] <fichier>
$ tail [options] <fichier>
```

Rappelons que la commande suivante affiche les 15 premières lignes du fichier `/var/log/syslog` (le résultat n'est pas affiché pour ne pas encombrer le texte) :

```
$ head -n 15 /var/log/syslog
```

L'utilisation d'une commande en tant que filtre se fait le plus souvent de la manière suivante (pour le même résultat) :

```
$ cat /var/log/syslog | head -n 15
```

De façon similaire, la commande suivante affiche les 15 dernières lignes du fichier `/var/log/syslog` :

```
$ cat /var/log/syslog | tail -n 15
```

✍ **Challenge C32Q1**

Entête

2.2 La commande `split`

La commande `split` permet de découper un fichier en plusieurs morceaux. Plus précisément, la commande `split` divise un fichier en plusieurs fichiers (si possible de même taille). Les fichiers créés par la commande `split` seront nommés `PREFIXEaa`, `PREFIXEab`, ..., la chaîne `PREFIXE` étant donnée en argument de la commande. L'exemple suivant permet de diviser le fichier `/var/log/syslog` en plusieurs fichiers de 22 octets, avec l'option `-b 22`, qui seront appelés : `log_aa`, `log_ab`, ...

```
split -b 22 /var/log/syslog log_
```

La commande suivante effectue la même division mais en séparant en fichiers ayant le même nombre de lignes, dans notre cas 30, avec l'option `-l 30`.

```
split -l 30 /var/log/syslog log_
```

3 Modification de l'affichage d'un fichier à l'écran

La commande `sort` permet d'afficher la sortie standard ou le contenu d'un fichier en triant les lignes. On peut choisir les colonnes qui seront les clefs du tri. Cette commande ne supprime pas d'information : en sortie toutes les colonnes seront présentes et le contenu de chaque ligne sera inchangé.

Utilisée sans option, la commande `sort` trie le fichier donné en argument dans l'ordre alphabétique. Prenons comme exemple le fichier suivant nommé `tri.txt` :

```
$ cat tri.txt
21 Béatrice 1 AA
3 Belle 2 BB
11 Alice 2 CC
```

```
$ sort tri.txt
11 Alice 2 CC
21 Béatrice 1 AA
3 Belle 2 BB
```

Notons ici que la commande précédente pourrait s'écrire : `cat tri.txt | sort`. Le tri réalisé ci-dessus se fait par ordre alphabétique. Comme vous le savez, dans l'ordre alphabétique 1 et 2 sont avant 3, c'est pourquoi ici la ligne qui commence par 3 se retrouve en dernière position. Pour changer ça et effectuer un tri numérique il faut utiliser l'option `-n`.

```
$ cat tri.txt | sort -n
3 Belle 2 BB
11 Alice 2 CC
21 Béatrice 1 AA
```

On peut aussi trier par ordre alphabétique sur la deuxième colonne, qui dans notre exemple correspond à la colonne des prénoms. Dans la commande `sort` on spécifie le numéro de la colonne avec l'option `-k`.

```
$ cat tri.txt | sort -k 2
11 Alice 2 CC
3 Belle 2 BB
21 Béatrice 1 AA
```

L'ordre des deux dernières lignes s'explique par le fait que dans l'ordre alphabétique de l'ordinateur `e` est avant `é`. On peut combiner les options `-n` et `-k` si on veut trier sur la troisième colonne de manière numérique.

```
$ cat tri.txt | sort -nk 3
21 Béatrice 1 AA
11 Alice 2 CC
3 Belle 2 BB
```

On peut aussi trier sur plusieurs colonnes. Par exemple, dans notre cas, on peut d'abord trier de manière numérique sur la colonne 3 et puis trier de manière alphabétique sur la colonne 4.

```
$ cat tri.txt | sort -nk 3 -k 4
21 Béatrice 1 AA
3 Belle 2 BB
11 Alice 2 CC
```

Les autres options usuelles de la commande `sort` sont `-t` pour changer le séparateur de colonne, et `-r` pour inverser l'ordre. Pour plus d'options, vous pouvez consulter la page de manuel de `sort`.

✍ **Challenge C32Q2**

Sont-elles ordonnées ?

4 Extraction d'information

La commande `cut` permet d'extraire des colonnes d'un fichier. Par exemple la commande `cut -c1-2 tri.txt` extrait les deux premiers caractères du fichier *tri.txt*, alors que `cut -f2,5 tri.txt` extrait les seconde et cinquième colonnes. Faites attention au fait que le séparateur par défaut est la tabulation, ce qui est différent du caractère espace. Le séparateur peut être modifié avec l'utilisation de l'option `-d`.

Considérons le fichier `extr.txt` suivant :

```
$ cat extr.txt
11 Alice 2 ; CC
21 Béatrice 1 ; AA
3 Belle 2 ; BB
```

Pour extraire la deuxième colonne contenant les prénoms, on utilise la commande suivante. Attention, il faut redéfinir le séparateur de colonne pour que ce soit une espace et non une tabulation avec l'option `-d " "`. L'option `-f 2` permet de sélectionner la seconde colonne.

```
$ cut -d " " -f 2 extr.txt
Alice
Béatrice
Belle
```

Notez à nouveau que la commande précédente, comme toutes les commandes de filtres, peut aussi s'écrire : `cat extr.txt | cut -d " " -f 2`

Pour extraire les deuxième et cinquième colonnes on utilise la commande suivante :

```
$ cat extr.txt | cut -d " " -f 2,5
Alice CC
Béatrice AA
Belle BB
```

Pour extraire les trois premières colonnes, on peut utiliser la commande précédente en y listant toutes les colonnes ou, dans le cas de notre fichier, changer le séparateur en caractère `';'` et extraire la première colonne en fonction de ce séparateur.

```
$ cat extr.txt | cut -d ";" -f 1
11 Alice 2
21 Béatrice 1
3 Belle 2
```

Il est aussi possible de spécifier un ensemble de colonnes. Par exemple, avec l'option `-f 1-3`, on peut extraire les colonnes 1 à 3 (soit les colonnes 1, 2 et 3). On peut combiner la sélection des colonnes avec l'écriture `-f 1-3,5,7` qui sélectionnera les colonnes 1, 2, 3,5 ,7. Pour sélectionner la colonne 3 et toutes celles qui la suivent on peut utiliser l'écriture `-f 3-`.

✍ **Challenge C32Q3**

Comment s'appellent-elles ?

5 Assemblage

5.1 La commande `cat`

La commande `cat` a été vue dans les activités précédentes. Elle permet de concaténer plusieurs fichiers les uns à la suite des autres. Les fichiers concaténés par la commande `cat` sont affichés sur la sortie standard. Supposons que nous ayons deux fichiers `low.txt` et `high.txt`. Le contenu de chaque fichier est le suivant :

```
$ cat low.txt
1
2
3

$ cat high.txt
4
5
6
```

```
$ cat low.txt high.txt
1
2
3
4
5
6
```

5.2 La commande paste

Contrairement à la commande **cut** qui permet d'extraire des informations d'un fichier ou de l'entrée standard, la commande **paste** permet de fusionner les colonnes contenues dans plusieurs fichiers. Attention la commande **paste** n'est pas présente dans la Weblinux.

La commande **paste** permet de réunir les deux fichiers comme le montre l'exemple suivant. Attention, l'ordre dans lequel les fichiers sont donnés en argument est important.

```
$ paste low.txt high.txt
1 4
2 5
3 6
```

L'option **-d** permet de modifier le caractère de séparation qui par défaut est la tabulation.

```
$ paste -d : low.txt high.txt
1:4
2:5
3:6
```

5.3 La commande join

La commande **join** est similaire à la commande **paste** car elle permet de mixer plusieurs informations provenant de fichiers ou de l'entrée standard. La commande **join** permet de joindre des fichiers en fonction de mots clés. Prenons l'exemple suivant :

```
$ cat nom.txt
BROOKS Belle xx
JACK Alice 10
DANIEL Beatrice 1
```

```
$ cat niveau.txt
Alice bash guru
Belle unknown status
Beatrice bash newbie
```

Pour lier les deux fichiers en fonction des prénoms, on utilise la commande suivante :

```
$ join -1 2 -2 1 nom.txt niveau.txt
Alice JACK 10 bash guru
Belle BROOKS xx unknown status
Beatrice DANIEL 1 bash newbie
```

L'option `-1 2` signifie que dans le premier fichier, soit dans notre cas *nom.txt*, le champ définissant la relation est la deuxième colonne. De même l'option `-2 1` signifie que dans le second fichier, dans notre cas *niveau.txt*, le champ définissant la relation est la première colonne. Attention au fait que dans la sortie de la commande, chaque ligne commence par la clef de relation.

6 Modification du contenu

Aucune des commandes précédentes ne supprime ni ne modifie une information spécifique. Dans cette section nous allons voir deux commandes simples permettant de modifier le contenu d'un fichier ou de l'entrée standard.

6.1 La commande `uniq`

La commande **`uniq`** permet de supprimer des lignes adjacentes identiques dans un fichier. Supposons que nous ayons le fichier suivant, appelé **`double.txt`** :

```
$ cat double.txt
AAAA
AAAA
BBBB
CCCC
CCCC
DDDD
AAAA
```

Le résultat de la commande **`uniq`** sur ce fichier sera le suivant :

```
$ uniq double.txt
AAAA
BBBB
CCCC
DDDD
AAAA
```

Les options les plus communes de la commande **`uniq`** sont **`-d`**, qui affichera seulement les lignes ayant des doublons adjacents, et **`-u`**, qui affichera seulement les lignes n'ayant pas de doublons adjacents.

```
$ uniq -d double.txt
AAAA
```

```
CCCC
```

```
$ uniq -u double.txt
BBBB
DDDD
AAAA
```

Les options `-f` (resp. `-s`) permettent d'ignorer un certain nombre de champs (resp. de caractères) en début de chaque ligne avant d'effectuer la suppression de doublons.

L'option `-c` permet à la commande `uniq` d'afficher le nombre de doublons adjacents rencontrés. Par exemple :

```
$ uniq -c double.txt
   2 AAAA
   1 BBBB
   2 CCCC
   1 DDDD
   1 AAAA
```

6.2 La commande `tr`

La commande `tr` (*translate*) remplace une liste de caractères par une autre. Supposons que l'on souhaite changer les majuscules en minuscules en utilisant la commande `tr`. Nous avons le fichier `slogan.txt` suivant :

```
$ cat slogan.txt
Live free or Die UNIX
```

Pour changer toutes les majuscules en minuscules, on utilise :

```
$ tr "[A-Z]" "[a-z]" < slogan.txt
live free or die unix
```

Dans cette commande, `"[A-Z]"` correspond à toutes les lettres majuscules et `"[a-z]"` correspond à toutes les lettres minuscules. Dans ce cas `'A'` sera remplacé par `'a'`, `'B'` par `'b'` etc. On peut aussi choisir de remplacer `'f'` par `'F'` et `'U'` par `'*'`. Pour cela il faut écrire la commande suivante :

```
$ tr "Uf" "*F" < slogan.txt
Live Free or Die *NIX
```

7 Meta-information : `wc`

La commande `wc` (*Word Count*) est un peu particulière pour cette section car elle ne permet pas de transformer, d'extraire ou de mixer des informations d'un fichier texte ou de l'entrée standard. La commande `wc` fournit des méta-informations sur un fichier ou sur l'entrée standard. `wc` affiche le nombre de lignes, mots et octets contenus dans les fichiers dont les noms sont donnés en argument.

Supposons que nous avons le fichier suivant :

```
$ cat slogan4.txt
Live Free or Die UNIX
Live Free or Die UNIX
Live Free or Die UNIX
Live Free or Die UNIX
```

La commande `wc` avec comme seul argument le nom du ficher produit la sortie suivante :

```
$ wc slogan4.txt
       4      20      88 slogan4.txt
```

La commande `wc` s'utilise aussi comme filtre avec la syntaxe suivante, qui produit le même résultat :

```
$ cat slogan4.txt | wc
       4      20      88
```

On obtient ici `4` qui est le nombre de ligne, `20` le nombre de mots et `88` le nombre de caractères. Pour obtenir seulement le nombre de lignes, la commande est `wc -l`. Pour obtenir seulement le nombre de mots la commande est : `wc -w`. Pour obtenir seulement le nombre de caractères la commande est : `wc -m`. Enfin, la commande `wc -c` permet d'obtenir le nombre d'octets.

✍ **Challenge C32Q4**

Combien sont-elles ?

8 Pour aller plus loin

Les exercices suivants vous permettent de vous familiariser avec les commandes vues dans ce chapitre. Les exercices utilisent les fichiers de la Weblinux.

Exercice 3.2.1: Pour aller plus loin.
Placez-vous dans le répertoire `cd /home/alice/Sequence3/imdb`. Dans ce répertoire, il y a des sous-répertoires nommés en fonction de nom de magazines américains célèbres. Chaque magazine fournit deux sous-répertoires `notes` et `photos`. Dans le répertoire `notes`, il y a un fichier `actress.csv` qui contient une liste d'actrices célèbres de la forme suivante : `Nom:Prénom:Note`. La note représente le vote des utilisateurs pour chaque magazine.

1. Alice voudrait classer dans l'ordre alphabétique du prénom les actrices se trouvant dans le fichier `actress` du sous-répertoire `gq` et afficher seulement les 15 premiers de cette liste.
2. Alice voudrait, dans le fichier `actress` du sous-répertoire `hollywood/notes`, afficher seulement le nom et le prénom des actrices sous la forme : « Nom Prénom ». (Notez le changement de séparateur).
3. Alice veut savoir quelles sont les actrices présentes uniquement dans la notation de `instyle/notes` ou uniquement dans la notation de `people/notes` et pas dans les deux en même temps.

Solution page 160

9 Conclusion

Dans cette activité nous avons vu comment modifier et transformer l'affichage de fichier texte ou de l'entrée standard. Les commandes vues dans cette section sont les outils de base permettant de traiter les fichiers texte de notre système d'exploitation afin d'en extraire les informations pertinentes.

Solutions des exercices

Solution de l'exercice 3.2.1 page 158:

1. `cat gq/notes/actress.csv | sort -k2 -t: | head -n 15`
2. `cat hollywood/notes/actress.csv | cut -d: -f1,2 | tr ":" " "`
3. `cat instyle/notes/actress.csv people/notes/actress.csv | sort | cut -d: -f1,2 | uniq -u`

Activité 3.3

Filtres puissants

1 Introduction

Vous êtes maintenant bien plus à l'aise avec votre ligne de commande. Vous gagnez en rapidité et en efficacité. Vous avez un certain nombre de commandes dans votre boite à outils avec lesquels vous arrivez à exploiter votre système d'exploitation. Il est donc temps pour vous de passer à la vitesse supérieure et de maîtriser toute la puissance de votre système.

Pour rappel, la philosophie des commandes UNIX dont les systèmes Linux ont hérité est : *Chaque commande fait une seule chose et le fait bien.* Cette section va introduire des filtres puissants par opposition aux filtres simples vus dans l'activité précédente. Les commandes que vous allez voir dans cette activité sont des commandes (basiques) qui font toute la puissance de la ligne de commande [1].

L'ensemble des options des commandes que nous allons voir dans cette activité ne seront pas toutes traitées. Nous en introduirons quelques unes pour avoir un avant-goût de leur puissance. En effet, ces commandes prises une à une peuvent faire l'objet d'une activité entière, voire même d'une séquence ou d'un MOOC entier !

2 La commande `find`

2.1 Introduction

Trouver un fichier dans son arborescence est l'une des tâches les plus fréquentes que l'on effectue devant un ordinateur. Jusqu'ici, pour trouver un fichier, vous vous servez de la commande `ls`. Cependant, cette commande ne vous permet pas d'explorer simplement une arborescence à la recherche d'un fichier ou d'un groupe de fichier précis. L'utilisation de l'option `-R` de la commande `ls` donne une liste récursive complète (par opposition à filtrée) de tous les fichiers et répertoires de l'arborescence sur laquelle est lancée la commande.

Une autre commande est bien plus adaptée pour cette tâche de recherche : la commande `find`. Cette commande va retrouver dans un répertoire, ou une liste de répertoires, un fichier ou un ensemble de fichiers possédant certaines caractéristiques comme : le nom, les droits, les dates, la taille, etc. ou satisfaisant à une expression donnée en argument.

La commande `find` possède beaucoup d'options dont les principales sont :

1. C'est l'avis subjectif de l'auteur

— `-name fichier` où fichier est le nom du fichier à trouver. Ce nom peut comporter les caractères utilisés pour la substitution de nom de fichier.. L'option `-iname fichier` indique de ne pas être sensible à la casse.
— `-perm nombre` où *nombre* équivaut aux droits d'accès.
— `-user nom` où *nom* correspond à un nom d'utilisateur et le résultat de la recherche listera tous les fichiers appartenant à cet utilisateur.
— `-size n` où *n* correspond à la taille du fichier
— `-exec cmd` où *cmd* correspond à une commande à exécuter sur chacun des fichiers trouvés.
— `-mtime n` où *n* correspond à une durée en jours depuis les dernières modifications.

Cette liste d'option n'est pas exhaustive, la page de manuel de la commande `find` donne la liste complète et, en fonction des systèmes, des exemples d'utilisation.

2.2 Exemples simples

Nous donnons ci-dessous quelques exemples courants de l'utilisation de la commande `find`.

```
$ find /etc /bin -name "a*"
```

Liste toutes les entités de `/etc` et `/bin` dont le nom commence par la lettre `'a'`. Les guillemets sont fondamentaux afin que le shell n'interprète pas le motif de recherche à la place de la commande `find`.

```
$ find . -iname "d*" -type d
```

Liste tous les répertoires (option `-type d`) contenus dans le répertoire courant (option `.`) dont le nom commence par les caractères 'd' ou 'D' (option `-iname`). Les valeurs pouvant être prises par l'option `-type` sont : `d` pour les répertoires, `l` pour les liens symboliques et `f` pour les fichiers.

```
$ find $HOME -mtime -7
```

Liste toutes les entités du répertoire personnel qui ont été modifiées il y a moins de 7 jours (de maintenant à -6 jours). L'option `-mtime +7` avec un nombre signé positivement indique de retrouver les fichiers modifiés il y a plus de 7 jours (de 8 jours à plus).

```
$ find . -size +1M
```

Liste toutes les entités du répertoire courant dont la taille dépasse le Mébioctet (Mio) [2]. L'option `-size -1G` listera les entités de moins de 1 Gibioctet (Gio). Note : Si le signes "-" ou "+" n'est pas présent avant la taille, `find` cherche le fichier exactement à la taille donnée.

```
$ find $HOME ! -user $LOGNAME
```

Liste toutes les entités de votre répertoire personnel dont vous n'êtes pas le propriétaire, car l'option `!` exprime la négation.

2.3 ET et OU avec la commande `find`

Il est possible de combiner les critères avec la commande `find`. Il suffit de saisir toutes les options. Attention, néanmoins, dans ce cas, ces critères sont pris comme des intersections (ET). Par exemple :

```
$ find . -size +15M -size -1000M
```

2. soit 1024*1024 octets

Liste toutes les entités du répertoire courant dont la taille dépasse quinze Mébioctets mais dont la taille est aussi inférieure à un Gibioctet. Note : l'unité des options `-size` doit être la même.

L'option `-o` de la commande `find` rend la recherche sur des unions de critères (OU).

```
$ find . -name "*.py" -o -name "*.cpp"
```

Liste toutes les entités du répertoire courant dont le nom se termine par `.py` ou par `.cpp`.

La combinaison des unions et des intersections est aussi possible.

```
$ find . -name "*.py" -size +10k -o -iname "*.jpg" -size +10M
```

Liste dans le répertoire courant les fichiers dont le nom se termine par `.py` et d'une taille supérieure à dix kilo octet, ou les fichiers dont le nom se termine par `.jpg` avec une taille supérieure à dix Mébioctets.

2.4 L'option -exec

Outre l'affichage, une commande peut être appliquée au résultat de la recherche obtenue par la commande `find`. Ceci s'effectue en utilisant l'option `-exec`. Cette option utilise une syntaxe particulière. Une ligne du résultat de la recherche se note par la chaîne de caractères `{}`. Aussi `{}` est remplacée par le nom du fichier courant là ou elle apparait dans l'argument de la commande à exécuter. L'option `-exec` se termine par le caractère `';'`. Ce caractère est aussi le caractère de séparateur de commandes pour le shell. Pour empêcher son interprétation par le shell, le point virgule doit être inhibé soit par l'inhibition de caractère (`'\'`) ou par l'inhibition totale (`' '`).

Voyons cela avec cet exemple :

```
$ find . -iname "*.jpg" -exec echo "--  {}  ++" \;  ; echo $?
```

Chaque nom de fichier trouvé est préfixé par `'--'` et suffixé par `'++'`. Le premier caractère `';'` indique la fin de la commande `echo`. Le second caractère `';'` est le séparateur de commandes du shell. La seconde commande de la ligne de commande affiche le code retour de la dernière commande exécutée par l'option `-exec`. On peut enchaîner des actions avec plusieurs `-exec` et utiliser plusieurs fois `{}` dans la même commande :

```
$ find . -name "*.txt" -exec wc -l {} ';' -exec cp {} {}.bak ';'
```

Trouve tous les fichiers dont le nom se termine par `.txt`, puis exécute la commande `wc -l` sur les entités trouvées (représentée par `{}`) et ensuite copie les entités trouvées dans des fichiers de même nom avec l'extension `{}.bak`.
On peut aussi enchaîner la commande `find` avec une autre commande :

```
$ find . -name "*.txt" -exec cat {} ';' | wc -l
```

Trouve tous les fichiers dont le nom se termine par `.txt` et compte le nombre de lignes au total pour les fichiers trouvés

✍ **Challenge C33Q1**

Angelina Jolie

3 La commande grep

3.1 Utilisation classique

La commande **find** vue dans la section précédente permet de trouver des fichiers avec certaines caractéristiques, mais ne permet pas de rechercher dans le contenu du fichier. La commande **grep** affiche toutes les lignes des fichiers données en argument, ou sur l'entrée standard, contenant une chaîne de caractères elle aussi donnée en argument. Prenons le fichier **slogan.txt** suivant comme exemple :

```
$ cat slogan.txt
There is nothing UNIX can't buy
Give that man a Bash
Proudly powered by linux
Crunch All you want. We'll make Bash
UNIX or nothing
Kids are stronger with linux
If you really want to know, look into linux
We're always low UNIX
Pleasing linux the world over
Linux inside
Live free or die UNIX
```

Voici un exemple d'utilisation de la commande **grep** sur ce fichier :

```
$ grep UNIX slogan.txt
There is nothing UNIX can't buy
UNIX or nothing
We're always low UNIX
Live free or die UNIX
```

Dans cet exemple, on cherche dans le fichier **slogan.txt** toutes les lignes contenant le mot 'UNIX'. Ici nous ne donnons qu'un seul fichier en argument mais il est possible d'en donner plusieurs.

```
$ grep UNIX fichier1.txt fichier2.txt
```

Les options de la commande les plus utilisées sont **-v** pour inverser la sélection, **-n** pour afficher le numéro de ligne et **-c** pour donner le nombre de lignes trouvées.

```
$ grep -v UNIX slogan.txt
Give that man a Bash
Proudly powered by linux
Crunch All you want. We'll make Bash
Kids are stronger with linux
If you really want to know, look into linux
Pleasing linux the world over
Linux inside
```

```
$ grep -vn UNIX slogan.txt
2:Give that man a Bash
3:Proudly powered by linux
4:Crunch All you want. We'll make Bash
6:Kids are stronger with linux
7:If you really want to know, look into linux
```

```
9:Pleasing linux the world over
10:Linux inside
```

```
$ grep -vc UNIX slogan.txt
7
```

Les motifs de recherche utilisés par la commande `grep` sont les expressions régulières. Il existe des livres entiers traitant des expressions régulières. Ce sont des chaînes syntaxiques qui donnent tout leur sens aux commandes comme `grep`. Les expressions régulières utilisées avec la commande `grep` permettent par exemple de rechercher des expressions complexes comme : trouver les lignes contenant les mots commençant par la chaîne de caractères 'th' ; les lignes contenant les mots de 4 lettres se terminant par 'w' ; etc. Nous n'allons pas examiner ici toutes les possibilités des expressions régulières, mais seulement donner quelques exemples basiques. Par ailleurs, la syntaxe utilisée pour les expressions régulières est présentée dans le manuel en ligne de `re_format` ou `regex` selon le système Unix.

L'accent circonflexe (`'^'`) signifie de chercher une expression en début de ligne :

```
$ grep '^Li' slogan.txt
Linux inside
Live free or die UNIX
```

Avec la commande précédente, on liste toutes les lignes commençant par la chaîne `Li`. Notez que l'expression régulière est placée entre guillemets simples afin d'appliquer une inhibition totale. En effet, il est plus prudent d'empêcher le shell de faire des substitutions en traitant les caractères spéciaux des expressions régulières. À l'inverse, du caractère (`'^'`), le caractère `$` indique de rechercher une expression en fin de ligne.

```
$ grep 'ux$' slogan.txt
Proudly powered by linux
Kids are stronger with linux
If you really want to know, look into linux
```

L'exemple liste toutes les lignes se terminant par 'ux'. Avec les crochets un ensemble de caractères est recherché dans un fichier. La commande suivante permet d'afficher toutes les lignes commençant par 'K' ou commençant par 'U'.

```
$ grep '^[KU]' slogan.txt
UNIX or nothing
Kids are stronger with linux
```

Les crochets servent aussi de définir une plage de caractères. Dans l'exemple suivant, toutes les lignes commençant par un caractère entre 'K' et 'U' sont affichées.

```
$ grep '^[K-U]' slogan.txt
There is nothing UNIX can't buy
Proudly powered by linux
UNIX or nothing
Kids are stronger with linux
Pleasing linux the world over
Linux inside
Live free or die UNIX
```

3.2 Options courantes de la commande grep

La commande **grep** possède plusieurs options.

— Dans les exemples précédents, nous avons vu que la chaîne de caractères recherchée est sensible à la casse. Pour enlever cette sensibilité, la commande **grep** propose l'option **-i** :

```
$ grep -i "unix" slogan.txt
There is nothing UNIX can't buy
UNIX or nothing
We're always low UNIX
Live free or die UNIX
```

— L'option **-w** limite la recherche à un mot entier (délimité par des espaces ou situé en début ou fin de phrase).

— Les options **-A, -B, -C** permettent d'afficher un certain nombre de ligne : avant **-B** (pour *before*), après **-A** (pour *after*) et autour **-C** (pour *center*). Par exemple, la commande suivante affichera les deux lignes situées avant, et trois lignes situées après la ou les lignes contenant le mot 'nothing' :

```
$ grep -B2 -A3 "nothing" slogan.txt
```

Alors que la commande suivante affichera les deux lignes situées avant, et les deux lignes situées après chaque ligne contenant le mot 'nothing' :

```
$ grep -C2 "nothing" slogan.txt
```

Il faut noter qu'il n'y a pas d'espace entre la lettre de l'option et la valeur indiquant le nombre de ligne.

— L'option **-E** demande l'utilisation de la version étendue de **grep**. Dans cette version, la syntaxe des expressions régulières est enrichie par de nouveaux caractères spéciaux. Citons le pipe '|' pour offrir une alternative. Par exemple, la commande suivante affiche les lignes contenant les mots UNIX ou Linux.

```
$ grep -E "UNIX|Linux" slogan.txt
[...]
```

Notez bien ici qu'il faudrait utiliser le caractère \ pour échapper le pipe | si les doubles quotes n'étaient pas utilisées.

✍ **Challenge C33Q2**

"Jennifer Aniston"

3.3 Expression régulière

Dans une expression régulière, le caractère '.' représente un caractère quelconque. L'exemple suivant liste toutes les lignes contenant une majuscule entre "A" et "Z" suivie d'un caractère quelconque et suivi d'un "v". Dans notre cas les mots correspondants sont 'Give' et 'Live'.

```
$ grep "[A-Z].[v]" slogan.txt
Give that man a Bash
Live free or die UNIX
```

Les accolades dans les expressions régulières permettent de spécifier des répétitions. Dans l'exemple suivant, on recherche les lignes contenant une lettre majuscule entre "R" et "Z", puis 4 caractères minuscules entre "a" et "z" puis une espace. Ici le mot correspondant est "There". Notez le caractère d'échappement \ devant les accolades ouvrante et fermante.

```
$ grep '[R-Z][a-z]\{4\} ' slogan.txt
There is nothing UNIX can't buy
```

Le tableau 1 synthétise tous les caractères spéciaux utilisés pour composer une expression régulière. A noter que la seconde partie du tableau indique les caractères spéciaux utilisés pour la version étendue de `grep`.

	Signification
`.`	un caractère quelconque
`*`	zéro ou plusieurs fois ce qui précède
`^`	l'expression régulière commence en début de ligne
`$`	l'expression régulière précède la fin de ligne
`\`	Inhibition de caractère
`[abf]`	un caractère pris dans un ensemble
`[a-f]`	un caractère pris dans l'intervalle
`[[:classe:]]`	un caractère pris dans une classe
`[^...]`	aucun caractère de l'ensemble ou de l'intervalle ou de la classe
`\{ \}`	un nombre de fois ce qui précède
`+`	un ou plusieurs fois ce qui précède
`?`	zéro ou une fois ce qui précède
`|`	alternative
`( )`	un groupe

Tableau 1 – Caractères spéciaux pour l'expression régulière.

3.4 Utilisation en filtre

La commande `grep` s'utilise souvent comme filtre. L'exemple suivant permet de rechercher "motif" dans les fichiers `f1`, `f2`, `f3`, et `f4`.

```
$ cat f1 f2 f3 f4 | grep "motif"
```

4 La commande awk

`awk` n'est pas à proprement parler une commande ou un filtre mais plutôt un langage de programmation. On le classe parmi les filtres programmables. `awk` est un outil de sélection et de manipulation de texte comme la commande `grep`. La syntaxe d'utilisation de la commande `awk` se présente sous la forme suivante :

```
$ awk 'commandes awk' fichier
```

`awk` traite les lignes du fichier une à une et de manière séquentielle. Dans `awk`, chaque ligne contient un certain nombre de champs séparés par un séparateur qui par défaut est une espace. Ce séparateur peut être modifié avec l'option `-F` de `awk`, ou encore en changeant la valeur de la variable `FS` interne à `awk`. Cette variable est modifiable dans la partie `'commandes awk'` de la syntaxe ci-dessus.

Dans tout ce qui suit, les variables qui vont apparaître dans la partie `'commandes awk'` sont des variables internes à `awk`. C'est-à-dire que ce ne sont pas des variables accessibles depuis le shell, elles ont seulement un sens bien spécifique dans les traitements réalisés par `awk`. Pour `awk` une ligne complète est contenue dans la variable `$0` et ses différents champs sont `$1`, `$2`, `$3`, ... numérotés de gauche à droite. Considérons le fichier `wcs.txt` suivant :

```
$ cat wcs.txt
Ada LOVELACE 1815 1852
Annie EASLEY 1933 2011
Grace HOPPER 1906 1992
```

La commande suivante permet d'inverser l'ordre prénom-nom en nom-prénom dans le fichier `wcs.txt`.

```
$ awk '{print $2 " " $1 " " $3 " " $4}' wcs.txt
LOVELACE Ada 1815 1852
EASLEY Annie 1933 2011
HOPPER Grace 1906 1992
```

Dans cette commande, les espaces entre les champs doivent être explicitement exprimées avec `" "`. Le mot clé `print` est un mot clé spécifique à `awk` qui permet d'afficher les variables. Comme spécifié précédemment, `awk` est un langage de programmation très riche, et nous n'entrerons pas dans les détails de celui-ci. Il permet de faire des calculs arithmétiques, de réaliser des tests logiques et contient de nombreuses fonctions (dont les fonctions mathématiques courantes).

Par exemple, `awk` permet facilement d'effectuer des calculs sur les champs. L'exemple suivant montre le calcul de l'âge et l'ajout d'un texte au moment de l'affichage du résultat. L'exemple modifie aussi le séparateur entre l'année de naissance et l'année du décès.

```
$ awk '{print $2 " " $1 " " $3 "-" $4 " age: " $4 - $3}' wcs.txt
LOVELACE Ada 1815-1852 age: 37
EASLEY Annie 1933-2011 age: 78
HOPPER Grace 1906-1992 age: 86
```

`awk` traite le ou les fichiers ligne par ligne, mais pas seulement. `awk` permet aussi de spécifier des blocs de pré-traitement et post-traitement. Ces blocs sont exécutés avant le traitement de la première ligne du ou des fichiers, et après le traitement de la dernière ligne. Pour cela, la syntaxe de `awk` à utiliser est la suivante (un bloc vide peut être omis) :

```
awk ' BEGIN { traitements débuts, avant lecture fichier}
            { traitement courants, ligne par ligne}
      END   {traitements fins, après parcours fichier} ' fichier.
```

L'exemple suivant montre l'utilisation de ces blocs.

```
awk ' BEGIN {print "Les informaticiennes :"}
            {print $2 " " $1 " " $3 "-" $4 " age: " $4 - $3}
      END   {print "Il y en a : " NR ". On en veut plus."}' wcs.txt
Les informaticiennes :
LOVELACE Ada 1815-1852 age: 37
EASLEY Annie 1933-2011 age: 78
HOPPER Grace 1906-1992 age: 86
Il y en a : 3. On en veut plus.
```

Le résultat de l'exemple précédent montre que l'on cherche à afficher, avant le traitement du fichier, la chaîne de caractères `Les informaticiennes :` présente dans le bloc `BEGIN`. Ensuite, à la fin du traitement nous avons cherché à afficher `Il y en a : " NR ". On en veut plus.`. La variable `NR` est une variable prédéfinie de `awk` qui contient le numéro de la ligne en cours d'analyse. Dans notre cas, puisque l'utilisation de cette variable se fait dans le bloc `END`, la valeur de `NR` correspond au numéro de la dernière ligne qui a été analysée. À l'affichage, cette variable est remplacée par sa valeur.

5 La commande sed

5.1 Utilisation de base

La commande `sed` (*stream editor*) est un éditeur de texte non-interactif, c'est-à-dire qu'il ne permet pas la saisie de texte mais plutôt d'éditer le contenu du fichier de façon automatisée. C'est en fait un filtre, dans sa version basique, qui reçoit un flux de textes sur son entrée standard et produit le résultat sur sa sortie standard. La commande `sed` peut traiter un flux de données de taille illimitée en utilisant très peu de mémoire. La commande `sed` est de ce fait un outil très rapide pour l'édition complexe de fichier.

La commande `sed` possède plusieurs options offrant une grande variété de fonctionnalités d'édition et de traitements. En effet, la commande `sed` peut utiliser les expressions régulières qui en font un filtre très puissant et dont les possibilités sont infinies. La commande `sed` lit les fichiers dont les noms sont indiqués en argument. Si aucun nom n'est indiqué, elle lit l'entrée standard. On lui indique les traitements à effectuer en utilisant l'option `-e` (la présence de cette option est facultative s'il n'y a qu'une seule directive de traitement). Par défaut les fichiers indiqués ne sont pas modifiés et le résultat de la transformation est écrit sur la sortie standard. Mais avec l'option `-i` on peut obtenir que le résultat soit écrit dans chacun des fichiers.

L'utilisation principale de la commande `sed` est l'opération de substitution de chaînes de caractères identifiées par une expression régulière. La forme syntaxique pour exprimer ce traitement est la suivante :

```
s/expression-reguliere/chaine/g
```

Cette directive de `sed` indique que l'on veut remplacer (s pour l'anglais *substitute*) les chaînes de caractères identifiées par `expression-reguliere` par la `chaine` donnée.

L'option `g` à la fin, indique que toutes lignes validant l'expression régulière seront traitées, sinon le traitement s'arrêtera après la première occurrence trouvée. La commande `sed` permet d'utiliser les expressions régulières étendues avec l'option `-r`, nous n'entrerons pas dans ce type d'expressions régulières ici. Pour utiliser les caractères spéciaux tels que `/` ou `&` dans les expressions ou la chaîne, il faut utiliser le caractère d'échappement anti-slash '\'. Il est possible d'effectuer plusieurs substitutions à la suite, elles seront alors traitées les unes après les autres et de gauche à droite.

Voici les exemples d'utilisation les plus courants :

```
$ sed 's/alice/bob/g' fichier1 fichier2
```

Lit le contenu des fichiers `fichier1` et `fichier2` et l'écrit sur la sortie standard en remplaçant tous les `alice` par `bob`.

Pour supprimer un mot il suffit de le remplacer par une chaîne vide. Par exemple, pour supprimer le mot `alice` la commande est :

```
$ sed 's/alice//g' fichier1 fichier2
```

Il est possible d'utiliser la commande `sed` comme filtre :

```
$ cat fichier | sed -e 's/a/A/g' -e 's/TA/ta/g'
```

La commande `cat` affiche le contenu du fichier `fichier` sur la sortie standard, qui est reprise par la commande `sed` (c'est le cas classique d'utilisation d'une commande filtre). Notez ici que nous avons utilisé l'option `-e` car la commande contient plusieurs directives de traitement. La première substitution change tous les `a` en `A` et la seconde tous les `TA` en `ta`. Ce qui signifie que si `fichier` contient

une ligne `a Table` , cette ligne sera transformée en `A table` . En effet, la première substitution transformera :

```
a Table --> A TAble
```

et la deuxième substitution agira sur le résultat de la première substitution :

```
A TAble --> A table
```

5.2 Options classiques

— Dans les exemples précédents, les substitutions sont effectuées sur l'ensemble des lignes. La syntaxe de substitution de la commande `sed` permet de spécifier les numéros des lignes sur lesquelles la substitution doit être effectuée. Par exemple, la commande suivante permet de remplacer 'UNIX' par '*nix' seulement sur les lignes 1 à 5 incluses :

```
$ sed '1,5s/UNIX/*nix/g' slogan.txt
```

— S'il est possible d'effectuer les substitutions seulement sur des lignes données en argument, il est aussi possible de spécifier les lignes en fonction d'un motif de recherche.

```
$ sed -e '/^C/s/Bash/BASH/g' slogan.txt
```

La commande précédente permet de remplacer 'Bash' par 'BASH' sur les lignes commençant par 'C'. La commande suivante permet de faire ce changement sur les lignes contenant le mot 'that'.

```
$ sed -e '/that/s/Bash/BASH/g' slogan.txt
```

— Jusqu'ici nous avons effectué des substitutions en utilisant un motif de recherche simple. Mais il est possible, comme pour la commande `grep`, d'utiliser des expressions régulières complexes. La commande suivante remplacera toutes les chaînes de caractères contenant 'L' et 'x' avec n'importe quel nombre de caractère entre ces deux lettres par "XXX".

```
$ sed -e 's/L.*x/XXX/g' slogan.txt
```

En remplaçant "XXX" par un "", la chaîne de caractères sera supprimée (remplacée par une chaîne vide).

— `sed` permet aussi de supprimer des lignes trouvées, au lieu de faire une substitution comme on l'a fait jusqu'ici. La commande suivante permet de supprimer, grâce à l'action `/d`, toutes les lignes commençant par `L` et se terminant par `e`.

```
$ sed -i '' -e '/^L.*e$/d' slogan.txt
```

L'option `-i "` (avec les deux quotes) permet d'effectuer les modifications directement dans le fichier `slogan.txt`. Attention selon les versions de la commande `sed` l'option est `-i "` (pour la famille Linux, GNU) ou `-i` (pour la famille Unix, BSD). Un autre exemple simple est la suppression de toutes les lignes contenant le mot `Linux`.

```
$ sed -e '/Linux/d' slogan.txt
```

✍ **Challenge C33Q3**

Changer pour mieux.

5.3 Options puissantes

La commande `sed` permet d'effectuer des modifications complexes sur les fichiers. Dans cette section nous en donnons quelques exemples.

— La commande suivante permet de préfixer chaque ligne par "Je dis : ".

```
sed -e 's/.*/Je dis: &/' slogan.txt
```

Dans cet exemple le caractère `&` indique à la commande `sed` d'insérer la chaîne de caractère trouvée par l'expression régulière. Dans notre cas `.*` permet de sélectionner toute une ligne (groupe de 0 ou plus de caractères).

— Mais la puissance de sed ne s'arrête pas là. La commande `sed` permet de déclarer des régions et de les réutiliser. Prenons le fichier `date.txt` suivant :

```
$ cat date.txt
2017-01-21
2019-11-09
2012-08-17
2003-15-13
```

Nous pouvons utiliser des régions en utilisant des parenthèses. Ainsi prenons la commande suivante dans laquelle les régions sont définies par les parenthèses `(.*)-(.*)-(.*)` (avec les caractère d'échappement). Chaque région est séparée par un tiret `-` comme dans le fichier `date.txt`. Une fois les régions définies, celles-ci peuvent être utilisées car un numéro leur est affecté. L'exemple suivant montre la transformation du fichier :

```
$ sed -e 's/\(.*\)-\(.*\)-\(.*\)/date: \3 \2 \1/' date.txt
date: 21 01 2017
date: 09 11 2019
date: 17 08 2012
date: 13 15 2003
```

Il existe beaucoup d'autres options et de possibilités offertes par la commande `sed`. Ici nous n'avons couvert que les cas les plus utilisés. Par exemple, en plus de la substitution, il y a l'ajout, l'insertion et la modification de lignes.

✍ **Challenge C33Q4**

On les veut toutes!!

6 Pour aller plus loin

Les exercices suivants vous permettent de vous familiariser avec les commandes vues dans ce chapitre. Les exercices utilisent les fichiers de la Weblinux.

Exercice 3.3.1: Placer le répertoire courant dans le répertoire `cd /home/alice/Sequence3/imdb`. Dans ce répertoire, il y a des sous-répertoires nommés en fonction de nom de magazines américains célèbres. Chaque magazine fournit deux sous-répertoires `notes` et `photos`. Dans le répertoire `notes`, il y a un fichier `actress.csv` qui contient une liste d'actrices célèbres de la forme suivante : `Nom:Prénom:Note`. La note représente le vote des utilisateurs pour chaque magazine.

1. Parmi toutes les actrices de tous les magazines, Alice voudrait savoir quelle est la pire note attribuée et à qui (au pluriel).
2. Alice se demande si les notes ne sont pas un peu biaisées. Elle se demande si il y a

une actrice qui peut être notée 99 par un ou plusieurs magazines et 0 par un ou plusieurs autres. Si c'est le cas, quel est/quels sont les noms de ces actrices qui ne font pas l'unanimité ?

Solution page 173

7 Conclusion

Dans cette activité, nous avons vu les filtres puissants. En effet, la commande `find` avec son option `-exec` peut être vue comme un filtre. Les commandes `sed, grep, awk` sont souvent utilisées après un pipe `|` comme des filtres puissants de la sortie standard.

Les commandes que nous avons vues ici (`find, grep, sed` et `awk`) sont, selon le rédacteur de ces lignes, les commandes les plus puissantes de linux car elles permettent vraiment de passer d'un stade de débutant à un stade d'utilisateur avancé sous linux. Maîtriser ces commandes, et les filtres puissants en général, font partie des fondamentaux de linux et vous venez d'acquérir ces fondamentaux. Les activités suivantes vous permettront de mieux exploiter votre ligne de commande, et vous n'éprouverez plus le besoin de la quitter.

Solutions des exercices

Solution de l'exercice 3.3.1 page 171:

1. ```
   find . -iname "actress.csv" -exec cat {} \; | sort -k3 -nr -t:
   ```

   Ici, nous regardons simplement la liste sans en isoler les actrices.

2. Il n'y en a qu'une et elle a joué dans « La vie est belle ».

   ```
   $ find . -iname "actress.csv" -exec cat {} \; | grep ":0\|:99" |
   sort | uniq | cut -f1,2 -d: | uniq -d
   ```
   ```

Activité 3.4

Effectuer des calculs numériques

1 Introduction

Le shell traite toutes les valeurs comme des données de type chaîne de caractères. Ainsi quand une expression arithmétique est formée, elle est traitée comme une chaîne de caractères et non comme une opération avec des nombres. C'est ce que montre l'exemple ci-dessous :

```
$ a=1
$ a=$a+1
$ echo $a
1+1
```

Cet exemple montre que la valeur de la variable `a` qui est le caractère "1" est complétée par la chaîne "+1" pour former la chaîne de caractères "1+1".

Pour effectuer des calculs numériques, le shell intègre une commande et des constructions spécifiques. Nous allons voir dans cette activité les différentes manières pour traiter les opérations arithmétiques. Ces opérations reposent sur des nombres entiers. Le traitement de nombres décimaux s'effectue par un calculateur dédié que nous verrons en fin d'activité.

Remarque : Typage de données Un type de donnée définit le type de valeurs que peut prendre une donnée. Le type de donnée sert à déterminer les opérations permises sur les données et comment les effectuer.

2 Opérations arithmétiques

Nous l'avons vu dans les activités précédentes : le shell Bash est le résultat d'une synthèse de différentes versions de shell. Il s'ensuit que le Bash n'est pas un langage avec une structure homogène. Il existe différentes façons d'effectuer la même chose. C'est le cas ici pour effectuer un calcul arithmétique. La première forme proposée repose sur la commande `expr`. Ensuite il est apparu une construction syntaxique dédiée que nous appellerons la substitution arithmétique. Les règles de l'évaluation d'une expression arithmétique ont été intégrées dans le shell. Ceci a permis d'offrir une commande interne au shell pour le traitement des expressions arithmétiques. Nous verrons ainsi la commande `let` et sa forme abrégée. La dernière forme proposée consiste à définir un type entier dans le shell pour lequel les expressions appliquées sur les variables du type entier sont évaluées avec les règles de l'évaluation arithmétique du shell.

2.1 Commande externe `expr`

La commande **`expr`** est une commande Unix qui prend pour arguments les termes d'une expression. Ainsi une addition s'écrit comme ceci :

```
$ expr 1 + 2
3
```

✍ **Challenge C34Q1**

Alice voyage en bus

Le résultat s'affiche sur la sortie standard. Il est important de noter que les termes et les opérateurs de l'expression sont les arguments de la commande **`expr`**. À ce titre, ils doivent donc être séparés par une espace. Cette contrainte rend l'utilisation de cette commande sujette à erreur.

```
$ expr 1+2
1+2
```

Cet exemple montre que "1+2" forme un mot et donc constitue un argument pour **`expr`**. La commande ne fait qu'afficher l'argument.

Dans le tableau 1, les opérateurs de la commande **`expr`** sont présentés dans l'ordre décroissant de priorité dans l'évaluation de l'expression.

Opérateur	Libellé
`\(` *expression* `\)`	parenthésage
`\*`	multiplication
`/`	division entière
`%`	reste de la division entière
`+`	addition
`-`	soustraction

Tableau 1 – Opérateurs de la commande **`expr`**.

Les caractères de parenthésage servent à ordonner les évaluations. L'addition étant moins prioritaire que la division (comme dans les mathématiques standards), l'expression $10 + 6/2$ sera évaluée par **`expr`** comme $6/2$ puis addition de 10 ce qui donne 13 :

```
$ expr 10 + 6 / 2
13
```

Pour faire la division d'une somme comme le montre l'exemple ci-dessous, il faut donc faire un parenthésage pour indiquer que l'addition est à faire avant la division :

```
$ expr \( 10 + 6 \) / 2
8
```

Les parenthèses sont des caractères spéciaux pour le shell. Il faut donc empêcher le shell de les traiter. C'est la raison pour laquelle, le caractère d'inhibition ('\') précède chaque parenthèse. Dans le tableau 1, vous pouvez remarquer que l'opérateur de multiplication est `\*` et non `*`. L'opérateur `*` est utilisé dans la substitution de nom de fichier, il indique tous les fichiers du répertoire courant. Cet opérateur, tout comme le parenthésage, doit être inhibé pour empêcher le shell de l'interpréter. C'est ce que montre l'exemple ci-dessous :

```
$ expr 2 * 2
expr: syntax error
$ expr 2 \* 2
4
```

✍ **Challenge C34Q2**

Les économies d'Alice

Comme nous l'avons indiqué les termes de l'expression sont les arguments de la commande `expr`. La conséquence c'est que ces termes sont évalués par le shell avant l'appel de la commande. L'évaluation peut consister à effectuer une substitution de variable afin de récupérer la valeur de la variable :

```
$ a=2
$ expr $a + 1
3
```

Ainsi la commande `expr` donne une méthode pour effectuer des calculs arithmétiques sur des variables. L'exemple ci-dessous montre l'incrémentation d'une variable. Le résultat de la commande `expr` est affecté à la variable `a` au moyen d'une substitution de commande `$( )`.

```
$ a=2
$ a=$(expr $a + 1)
$ echo $a
3
```

La commande `expr` est une commande historique de Unix. Elle possède une syntaxe de description des expressions qui lui est propre. Ceci oblige de retenir ses spécificités comme l'interprétation de certains caractères spéciaux du shell qu'il faut inhiber. Nous avons vu que le positionnement des espaces est important. Lorsqu'on utilise des variables il faut penser à faire précéder la variable par le caractère `$`. Ceci en fait une syntaxe fragile dans le sens où on peut vite faire une erreur d'écriture si on est distrait. Pour rendre la syntaxe moins fragile, le Bash propose une forme syntaxique dédiée pour les calculs arithmétiques. C'est ce que nous allons voir dans le paragraphe suivant. L'avantage de la commande `expr` est qu'elle ne dépend pas d'une version de shell. C'est une commande externe au shell, elle fonctionne donc quel que soit le shell utilisé. Aussi, elle reste un choix pertinent pour satisfaire des contraintes de portabilité.

2.2 Substitution arithmétique

Une substitution arithmétique consiste à évaluer une expression arithmétique et à la remplacer par le résultat de cette évaluation. Le format de la substitution arithmétique est :

```
$(( expression ))
```

Cette construction syntaxique du Bash présente de nombreuses améliorations par rapport à la commande `expr` qui rendent son usage plus simple pour l'utilisateur. Citons :

- Les termes de l'expression n'ont plus besoin d'être séparés par des espaces.
- L'usage d'une variable s'effectue sans la préfixer par le caractère `$`.
- Les caractères spéciaux du shell ne sont pas interprétés.
- La commande est interne au shell. Elle s'exécute dans le même processus que le shell lui-même.

Ainsi la multiplication de 2 variables dont le résultat est affecté à une troisième s'écrit de la manière suivante :

```
$ a=2 ; b=3
$ i=$((a*b))
$ echo $i
6
```

Cet exemple montre que l'écriture d'une expression arithmétique est bien plus simple avec cette construction qu'avec la commande `expr`.

Un autre avantage de cette construction porte sur les opérateurs et leur priorité d'évaluation. Ils sont les mêmes que pour le langage C. Ainsi avec la substitution arithmétique le shell propose une évaluation arithmétique bien plus flexible et riche que la commande `expr`. Le tableau 2 montre les opérateurs arithmétiques dans l'ordre de priorité décroissant d'évaluation.

Opérateurs	Libellés
`( )`	parenthésage
`++ --`	incrément, décrément
`!`	négation logique
`**`	puissance
`* / %`	multiplication, division, reste
`+ -`	addition, soustraction
`<= >= < > == !=`	comparaison
`&&`	et logique
`||`	ou logique
`= *= /= %= -= +=`	affectation

Tableau 2 – Opérateurs et leur priorité dans l'évaluation arithmétique du shell.

L'exemple ci-dessous montre un calcul de puissance

```
$ i=2
$ echo $((i**3))
8
```

La substitution arithmétique trouve ici son utilité à remplacer la commande `expr` en apportant des avantages significatifs par rapport à cette dernière.

Le tableau 2 indique qu'une expression évaluée par le shell peut être une affectation. C'est-à-dire qu'une expression peut être $i = 1 + 2$. Vérifions :

```
$ $((i=1+2))
-bash: 3: command not found
$ echo $i
3
```

L'expression est correcte, la variable `i` est correctement affectée. C'est la construction utilisée qui n'est pas bonne. Car la substitution arithmétique retourne une valeur au shell qui l'interprète comme une commande à exécuter. La commande 3 n'existe pas et produit donc une erreur. Nous allons voir dans le paragraphe suivant une autre commande du shell permettant d'évaluer une expression arithmétique selon les règles indiquées par le tableau 2, qui est plus adaptée à l'usage évoqué ici.

2.3 Commande interne `let`

Les expressions arithmétiques sont évaluées par la commande interne `let "expression"` qui s'abrège par la construction `((expression))`. Cette commande repose aussi sur les règles de l'évaluation arith-

métique du shell indiquées par le tableau 2, mais contrairement à la substitution arithmétique la commande `let` n'effectue aucune sortie. Ainsi l'expression peut être une affectation sans provoquer d'erreur :

```
$ ((i=1+2))
$ echo $i
3
```

Comme pour la substitution arithmétique, les variables n'ont pas besoin d'être préfixées par le caractère `$`. Il n'est pas non plus nécessaire d'appliquer des espaces entre les termes d'une expression. Par exemple, l'incrémentation d'une variable s'écrit :

```
$ a=10
$ ((a=a+1))
$ echo $a
11
```

Il y a plusieurs façons d'indiquer une incrémentation. L'exemple ci-dessous montre trois formes équivalentes :

```
1 $ let "a=a+1"
2 $ let "a+=1"
3 $ ((a++))
```

La première ligne de cet exemple utilise la commande `let` à la place de son abréviation (*(( ))*). Les doubles quotes visent à protéger l'expression de l'expansion des caractères spéciaux du shell. En effet, examinez les écritures suivantes :

```
$ let a=a*2
-bash: let: *: syntax error
$ let "a=a*2"
```

Pour en revenir à l'exemple de l'incrémentation, on retiendra souvent la forme la plus concise représentée par la dernière ligne : `((a++))`. La seconde ligne comporte la commande `let "a+=1"`. L'affectation précédée d'un opérateur indique que l'opération doit être appliquée à la valeur de la variable indiquée avant le signe égal, puis réaffectée à cette même variable. Aussi, nous savons ici que la valeur de la variable `a` est additionnée d'une unité. Cette écriture est utilisée à la place de `a++` pour effectuer des incréments supérieurs à 1.

```
$ a=2
$ let "a+=3"
$ echo $a
5
```

Ce qu'il faut retenir, c'est que cette commande offre une grande flexibilité et qu'elle est bien plus simple à utiliser que la commande `expr`. Comme elle reprend la syntaxe commune des langages de programmation et plus particulièrement celle du langage C, écrire une expression pour le shell ne demande plus de mémoriser les spécificités du shell dans l'utilisation des variables et des espaces. Cependant elle reste compatible avec les expressions utilisées avec la commande `expr`.

```
$ a=10
$ ((a = \( $a + 20 \) / 2 ))
$ echo $a
15
```

✍ **Challenge C34Q3**

Alice préfère le calcul sans expr

La commande `let` permet de faire plus que des calculs sur des entiers. Comme toutes les commandes des systèmes Unix, une fois exécutée la commande retourne un code d'état dans la variable `$?`. Si le code est à 0, cela indique une exécution correcte. Lorsque l'expression est une expression booléenne, le code retour prend la valeur 0 pour les cas où l'évaluation de l'expression a pour valeur vraie.

```
$ ((10<20))
$ echo $?
0
$ a=10
$ ((a==20))
$ echo $?
1
```

Nous reviendrons dans la séquence 4 sur les expressions et leur utilisation.

Attention de ne pas confondre la commande `(( ))` avec la substitution arithmétique `$(( ))`. La substitution remplace l'expression par le résultat de son évaluation, alors que la commande `(( ))` se comporte comme son nom l'indique comme une commande sur une ligne de commande. De plus elle n'affiche aucun résultat.

```
$ echo $( ((1+2)) ) #ici on capture avec $( ) l'affichage produit par la commande
   let

$ echo $((1+2))
3
```

2.4 Déclaration de variable de type entier

Comme nous l'avons indiqué en début de cette activité, toutes les valeurs manipulées par le shell sont vues comme des chaînes de caractères. Cependant, il peut être intéressant d'avoir des variables qui soient typées pour des entiers. Pour ces variables, les valeurs contenues le seront sous la forme d'une valeur entière. Avec des variables typées entier, il est possible d'effectuer des opérations arithmétiques sans avoir recours aux commandes `expr` ou `let`.

Pour typer une variable, il faut en faire la déclaration en spécifiant son type entier. Pour cela le Bash propose une commande interne `declare` qui se nomme aussi `typeset`. La syntaxe de la déclaration est la suivante :

```
  declare -i nom[=expression]
```

Avec la déclaration de la variable, il est possible de restreindre la variable à un type entier et de l'initialiser avec une expression arithmétique dont l'évaluation faite par le shell donnera sa valeur initiale :

```
$ declare -i a=3*2
$ echo $a
6
$ declare -i b
$ b=(a+2)/2
$ echo $b
```

```
4
```

Ainsi pour une variable typée entière, l'expression spécifiée dans une affectation est évaluée par le shell selon les règles de l'évaluation arithmétique que nous avons vues avec l'instruction `let` et la substitution arithmétique.

Outre une évaluation implicite (sans avoir recours à la commande `let`), la déclaration de variable assure que la valeur est toujours un entier. Si une chaîne de caractères non numérique est affectée, la conversion en valeur entière donnera la valeur 0.

```
$ declare -i a
$ a=essai
$ echo $a
0
```

Lorsqu'une variable de type entier est utilisée avec la commande `expr` et qu'elle a été affectée par un mot, une erreur à l'exécution est évitée :

```
$ a=$(expr $a + 1)
$ b=essai
$ b=$(expr $b + 1)
expr: non-numeric argument
```

La construction sur la base d'une déclaration est la forme la plus proche des langages de programmation. Elle permet de standardiser l'écriture des calculs arithmétiques pour le shell.

3 Calculs de nombres décimaux

Jusqu'ici, nous n'avons effectué des calculs que sur des nombres entiers. Dans l'utilisation courante d'un shell qui vise à faire des comptages, c'est bien suffisant. Mais s'il est nécessaire de faire des calculs scientifiques, il faut avoir recours à une commande dédiée.

On peut distinguer 2 commandes pour effectuer des calculs sur des nombres décimaux. La première commande est `awk` que vous avez découvert dans l'activité 3.3. La seconde commande est `bc` (*basic calculator*). Cette commande est l'équivalent d'une calculatrice programmable. Elle offre un langage de programmation pour réaliser des programmes de calculs. Elle peut s'utiliser en mode interactif en tapant simplement la commande `bc` :

```
$ bc -q
10/3
3
quit
$
```

L'option `-q` indique de ne pas afficher la bannière de démarrage. Dans cet exemple, nous avons démarré le calculateur puis saisi l'expression `10/3`. `bc` a affiché le résultat et nous avons quitté `bc` en saisissant la directive `quit`. Par défaut `bc` ne traite pas les décimales. Pour prendre en compte les décimales, il faut spécifier la précision par la directive `scale`. En indiquant `scale=10`, la commande `bc` codera les nombres sur une précision de 10 décimales.

```
$ bc -q
scale=10 ; 10/3
3.3333333333
quit
```

```
$
```

On notera que le séparateur de directive dans `bc` est le caractère point-virgule. Plusieurs directives peuvent ainsi être mises sur la même ligne. Le mode interactif signifie que l'interaction est faite avec la commande bc mais pas depuis le shell Bash. Pour effectuer des calculs à partir du shell, il faut utiliser `bc` comme un filtre. Il faut envoyer sur l'entrée standard de `bc` les instructions pour effectuer le calcul souhaité. `bc` donnera en sortie le résultat du calcul.

```
$ b=10
$ echo "scale=5 ; $b/3" | bc
3.33333
```

Cet exemple montre que l'expression à évaluer par `bc` doit être transmise à son entrée standard. Pour ce faire, la sortie de la commande `echo` est connectée à l'entrée de la commande `bc` via un pipe. Le résultat de `bc` est ensuite affiché sur la sortie standard.

✍ **Challenge C34Q4**

L'addition s'il vous plaît

4 Pour aller plus loin

Les exercices suivants vous permettent de vous familiariser avec les commandes vues dans ce chapitre. Les exercices utilisent les fichiers de la Weblinux.

Exercice 3.4.1: Placer le répertoire courant dans le répertoire `cd /home/alice/Sequence3/imdb`. Dans ce répertoire, il y a des sous-répertoires nommés en fonction de nom de magazines américains célèbres. Chaque magazine fournit deux sous-répertoires `notes` et `photos`. Dans le répertoire `notes`, il y a un fichier `actress.csv` qui contient une liste d'actrices célèbres de la forme suivante : `Nom:Prénom:Note`. La note représente le vote des utilisateurs pour chaque magazine.
Alice veut calculer la moyenne des notes attribuées à Julia Roberts par tous les magazines.

1. Pour cela, elle doit d'abord isoler toutes les notes et les mettre dans un fichier `/tmp/julia`.
2. Alice doit ensuite calculer, avec la commande wc, le nombre de notes obtenues par Julia Roberts qui se trouvent maintenant dans le fichier `/tmp/julia`, qu'elle va mettre dans la variable `count`
3. Alice doit ensuite calculer la somme totale des notes obtenues qui se trouvent maintenant dans le fichier `/tmp/julia`, qu'elle va mettre dans la variable `total`. Elle peut se servir pour cela de la commande `bc`.
4. Enfin, Alice calcule la moyenne de Julia Roberts en utilisant la commande `expr` (la valeur obtenue est un entier).

Solution page 184

5 Conclusion

Dans cette activité nous avons vu comment utiliser la commande de calculs arithmétiques `expr`. La syntaxe propre à cette commande la rend difficile à utiliser. Les erreurs d'écritures sont assez faciles

à faire. Le shell a alors été étendu avec des capacités d'évaluation des expressions arithmétiques. L'expression arithmétique du shell reprend la syntaxe du langage de programmation C qui est commune à beaucoup de langages de programmation. Avec cette capacité, le shell offre une syntaxe classique pour les expressions rendant leur écriture plus simple pour la personne ayant déjà des connaissances en programmation.

La première construction qui utilise cette capacité du shell à évaluer des expressions arithmétiques que nous avons vue est la substitution arithmétique. Elle s'utilise comme argument d'une commande. Ensuite nous avons vu qu'une expression arithmétique peut être évaluée comme une commande à part entière avec l'écriture `(( ))`. De plus, avec la commande `declare`, lorsqu'une commande d'affectation est exécutée l'expression est automatiquement évaluée par le shell si la variable affectée a été déclarée de type entier. Enfin, effectuer un calcul numérique avec des nombres décimaux repose sur la commande `bc`. À la différence de la commande `expr` pour laquelle les termes de l'expression sont passés sous forme d'arguments, avec `bc` l'expression doit être fournie sur son entrée standard. Elle s'utilise donc comme un filtre.

Solutions des exercices

Solution de l'exercice 3.4.1 page 182:
Pour information, Julia Roberts obtient une moyenne un peu en dessous de 50.

1. ```
   $ find . -iname "actress.csv" -exec cat {} \; | grep Roberts:Julia |
     cut -f 3 -d: > /tmp/julia
   ```

   Ici, nous mettons toutes les notes de Julia Roberts dans le fichier en ayant pris soin d'enlever le Nom et Prénom.
2. ```
   $ count=$(wc -l /tmp/julia | cut -f 1 -d " ")
   $ echo $count
   ```

   Ici on fait un `echo $count` pour vérifier le contenu de la variable.
3. ```
   $ total=$(echo $(cat /tmp/julia) | tr " " + | bc)
   $ echo $total
   ```

   Ici on fait un `echo $total` pour vérifier le contenu de la variable.
4. La moyenne se calcule simplement en divisant $total par $count.

   ```
   $ expr $total / $count
   ```

Avec la même procédure, vous pouvez calculer les moyennes des autres actrices et voir si elles sont mieux ou moins bien notées que Julia Roberts.
   ```

Activité 3.5

Archiver et compresser des données

1 Introduction

Votre ligne de commande est maintenant un outil que vous maîtrisez bien. Vous savez comment manipuler des fichiers et en extraire des informations, et vous savez effectuer des calculs. En somme, vous savez utiliser votre ligne de commande pour la plupart de vos tâches quotidiennes.

Actuellement, nous avons tous plusieurs appareils connectés en réseau. Les tâches effectuées sur chaque appareil sont différentes mais il arrive souvent que l'on veuille partager ou transférer des informations entre ces appareils ou même qu'on souhaite effectuer une action sur un appareil différent de celui qu'on utilise actuellement.

Le travail à distance peut facilement devenir complexe à cause de l'hétérogénéité des systèmes utilisés. Les mêmes logiciels doivent être présents sur tous les appareils et probablement avec les mêmes versions. L'avantage de la ligne de commande, et de Bash en particulier, c'est qu'une fois installées, les versions sont compatibles. De plus, contrairement à une interface graphique qui doit s'adapter à la taille de l'affichage, la ligne de commande ne nécessite aucune adaptation.

Opérer à distance est un domaine dans lequel la ligne de commande excelle. En effet, l'utilisation de la ligne de commande ne nécessite pas d'interface graphique très consommatrice en bande passante surtout sur les réseaux à faible débit. Malgré cette absence d'interface graphique, il est possible, comme vous avez pu le constater dans les activités précédentes, d'effectuer toutes les tâches classiques.

Dans cette activité, vous allez apprendre à archiver et compresser des fichiers ou dossiers, à transférer des fichiers depuis votre machine vers une machine distante ou inversement et à vous connecter à une machine distante pour effectuer un travail à distance.

2 La commande `tar`

La commande `tar` (*Tape ARchiving*) sert à transformer une hiérarchie de fichiers et/ou de dossiers en une archive prenant la forme d'un seul fichier. En faisant l'archive, `tar` conserve les attributs comme les droits, le propriétaire, le groupe des fichiers et des répertoires. De plus, il maintient également les éventuels liens symboliques. La commande `tar` est réversible : elle sert aussi bien à la création de l'archive qu'à l'extraction des fichiers de l'archive. Il faut noter que la création d'une archive ne compresse pas (réduire la taille) les données. Une archive est donc un regroupement de fichiers. Dans le langage technique, le fichier archive est qualifié de *tarball*. De nos jours, une archive se crée sur le disque sous la forme d'un fichier et non sur une bande magnétique (comme le suggère le nom de la commande).

Les utilisations principales de la commande `tar` sont :

La sauvegarde Il est plus facile de sauvegarder un seul fichier plutôt qu'une hiérarchie de répertoires. De plus, il est parfois complexe de stocker la hiérarchie de fichiers sur un système de fichiers ne supportant pas tous les noms et attributs de fichiers connus par Unix. Par exemple une clef USB formatée avec le système FAT n'est pas sensible à la casse. Transformer une hiérarchie en un fichier permet de s'affranchir de ces problèmes.

Le transfert Transférer une hiérarchie de fichiers en un bloc plutôt qu'en transférant de multiples fichiers permet de s'assurer plus facilement d'un transfert correct. Vérifier l'intégrité d'un fichier est beaucoup plus facile que de vérifier l'intégrité de toute une hiérarchie de fichiers ou de sous-répertoires.

Les opérations les plus courantes avec la commande `tar` sont :

— Création d'une archive :

```
tar cvf nom_a_creer.tar chemin_repertoire_existant
```

Le chemin vers le répertoire à archiver ne doit ni commencer par le caractère `'/'` ni par `'.'` ou `'..'` afin que l'on puisse choisir la destination lors de l'extraction.

✍ **Challenge C35Q1**

Seulement deux.

— Extraction d'une archive :

```
tar xvf nom_de_l_archive.tar
```

L'extraction est faite dans le répertoire courant.

— Lister le contenu d'une archive :

```
tar tvf nom_de_l_archive.tar
```

— Extraire seulement une partie d'une archive. Si votre archive contient par exemple le répertoire `rep`, vous pouvez n'extraire que ce répertoire :

```
tar xvf nom_de_l_archive.tar rep
```

En bref, les options courante de `tar` sont :

- `t` (*Table of content*) Afficher la liste du contenu d'une archive
- `x` (*eXtract*) Extraire du contenu d'une archive.
- `c` (*Create*) Créer une archive.
- `v` (*Verbose*) Afficher toutes les opérations effectuées par la commande et lister la hiérarchie de fichiers et sous-répertoires parcourus.
- `f` (*File*) Enregistrer le contenu de l'archive dans le fichier dont le nom suit l'option.
- `z` (*gZip*) Appliquer une compression à l'archive (voir la section suivante) (avec l'option `'c'` uniquement).

L'ancienneté de cette commande fait qu'il y a deux façons d'indiquer les options. L'ancienne consiste à donner les options sous forme d'un mot qui suit la commande `tar` (comme les exemples montrés ci-dessus). La moderne reprend la convention des options des autres commandes à savoir commencer l'argument de la commande par un tiret.

À propos de l'usage de l'option `f`, celle-ci indique que le nom de l'archive suit l'option. Il faut donc prendre soin de terminer les options par celle-ci. La commande ci-dessous archivera le répertoire `X` dans une archive nommé `v`.

```
tar -cfv X
```

Et en l'absence de l'option `f`, le contenu de l'archive s'affichera sur la sortie standard. En mode extraction ou affichage de la liste du contenu, l'option `z` n'est pas prise en compte. Dans la plupart des mises en oeuvre de la commande `tar`, la décompression est automatiquement effectuée quand une archive est lue.

✍ **Challenge C35Q2**

La confiance n'empêche pas le contrôle.

3 La compression et la transformation de données

3.1 La commande gzip

La commande `gzip` (*Gnu ZIP*) permet de compresser des fichiers pour réduire la place qu'ils prennent sur le disque. Cette compression est souvent utilisée avant le transfert du fichier à travers un réseau de communication pour réduire le temps de transfert. La commande `gzip` s'utilise de la manière suivante :

```
gzip monFichier
```

Le résultat de cette commande est la création d'un fichier `monFichier.gz` et la suppression du fichier `monFichier`. Il est possible de donner plusieurs noms de fichiers ou de compresser récursivement un répertoire sans en charger la hiérarchie. Il faut aussi noter que compresser plusieurs fois un fichier ou compresser un fichier déjà compressé ne permet pas de gagner plus de place sur le disque. Il est ainsi inutile de compresser un fichier `.gz, .mp3, .jpg, ...` car la taille économisée ne sera pas significative. Il existe d'autres compresseurs comme `bzip2, lzma, xz` mais ils ne sont pas tous systématiquement pré-installés sur les systèmes de type Unix.

tar et gzip

La commande `gzip` est aussi une option de la commande `tar`. Ainsi dans la commande
`tar czvf f.tgz f/`
l'option `z` demande de compresser directement l'archive créée. Dans notre cas, l'archive du répertoire `f/` sera compressée et sera dans le fichier `f.tgz`.

La commande de compression `gzip` s'utilise le plus couramment en combinaison avec la commande `tar`. La commande `tar` transforme une hiérarchie en un seul fichier. Ce fichier est une archive dont le nom est suffixé par l'extension `.tar` comme par exemple `archive.tar`. Puis cette archive est compressée avec la commande `gzip` pour obtenir le fichier suffixé par l'extension `.tar.gz` comme dans notre exemple `archive.tar.gz`. Ce dernier fichier est parfois renommé `archive.tgz` pour que le nom ne comporte qu'un seul point et qu'une seule extension.

Pour conserver le fichier original et en même temps produire un fichier compressé, on utilise l'option `-c` de la commande `gzip`. Attention au fait que cette option ne crée plus automatiquement le fichier résultat, mais redirige les données compressées vers la sortie standard. Aussi, il faut rediriger cette sortie vers le fichier à produire :

```
gzip -c monFichier > monFichier.gz
```

La commande précédente permet de conserver le fichier `monFichier` et de créer un fichier `monFichier.gz`. Enfin l'option `-v` (*Verbose*) affiche le nom et le pourcentage de réduction pour chaque fichier compressé. De manière générale, utiliser le mode verbeux est un bon réflexe à avoir pour vérifier que l'exécution de la commande s'effectue comme attendu.

✍ **Challenge C35Q3**

Make -riquiqui.

3.2 La commande gunzip

La commande `gunzip` (*GNU unzip*) permet de décompresser des fichiers d'extension `.gz`. Ainsi, la commande `gunzip` va créer un fichier `monFichier` et détruire le fichier `monFichier.gz`.

```
gunzip monFichier.gz
```

Pour conserver le fichier original et en même temps produire le contenu du fichier décompressé sur la sortie standard, on utilise l'option `-c`, comme pour la commande `gzip`. Avec cette option il faut rediriger la sortie standard vers le nom du fichier à produire :

```
gunzip -c monFichier.gz > monFichier
```

La commande précédente permet de conserver le fichier `monFichier.gz` et de créer un fichier `monFichier`.

✍ **Challenge C35Q4**

En même temps.

3.3 La commande zcat

La commande `zcat` prend comme argument des fichiers compressés et affiche leurs contenus décompressés. Ainsi la commande `zcat` affiche le fichier non compressé, sans modifier `monFichier.gz` ni créer `monFichier` sur le disque.

```
zcat monFichier.gz
```

Cette commande est utile si l'on veut vérifier le contenu d'un fichier sans pour autant le décompresser. La commande `zcat` est souvent considérée comme l'équivalent de la commande `gunzip -c`.

3.4 Les commandes uuencode et uudecode

Il existe plusieurs types de fichier sous linux. Les fichiers au format texte et les fichiers binaires. Il arrive que l'on veuille passer d'un format à un autre. En effet, sur certains canaux de transmission le transfert de fichier binaire n'est pas possible. Les commandes `uuencode` et `uudecode` permettent de passer d'un format binaire à un format ASCII et vice versa. Les commandes `uuencode` et `uudecode` s'utilisent de la manière suivante :

```
uuencode monFichier.xls > monFichier.txt
```

Cela transforme le fichier `monFichier.xls` en un fichier `monFichier.txt` en encodage texte, et il est ensuite possible de supprimer le fichier `monFichier.xls` s'il n'est plus utile.

```
uudecode monFichier.txt > monFichier.xls
```

La commande précédente fait l'opération inverse en retransformant le fichier `monFichier.txt` en sa version au format binaire sous le nom `monFichier.xls`.

4 Transférer les données

4.1 La commande scp

Dans les systèmes Unix, il est possible de copier des fichiers entre deux systèmes distants. La commande `scp` utilise le protocole SSH pour transférer les données. Cette commande nécessite un nom d'utilisateur et un mot de passe sur les deux systèmes (source et destination). Ces noms d'utilisateurs et mots de passe peuvent être différents. La syntaxe et les options de la commande `scp` sont très proches de celles de la commande `cp`. Contrairement à d'autres méthodes permettant de copier des fichiers depuis ou vers un système distant au travers du réseau, la commande `scp` sécurise le transfert avec l'utilisation de SSH, il sera donc impossible à une personne capturant le transfert sur le réseau de reconstituer le contenu du transfert.

```
scp alice@src:RepSrc/f1 bob@dest:RepDst/f2
```

La commande précédente permettra de copier le fichier `RepSrc/f1` se trouvant sur l'ordinateur `src` qui possède une utilisatrice `alice` vers un fichier `RepDst/f2` se trouvant sur l'ordinateur `dest` qui possède un utilisateur `bob`. Les valeurs de `src` et de `dest` peuvent par exemple être les adresses `IP` des ordinateurs ou leur nom de domaine (nous y reviendrons un peu plus loin).

L'utilisation la plus courante de la commande `scp` est la copie d'un fichier se trouvant sur l'ordinateur local vers un ordinateur distant.

```
scp /home/alice/f1 bob@dest:/home/bob/f1
```

La commande précédente permet de copier le fichier `/home/alice/f1` (fichier local) vers le fichier distant `/home/bob/f1` de l'utilisateur `bob` se trouvant sur la machine `dest`.

Secure Shell (SSH)

SSH est un protocole permettant d'effectuer des opérations sécurisées (utilisant la cryptographie) au travers d'un réseau. Le meilleur exemple d'utilisation de SSH est la connexion à distance sur un système.

Pour identifier une machine sur le réseau et/ou sur Internet, on utilise souvent une adresse. Cette adresse, appelé adresse IP (pour Internet Protocol) permet d'identifier de manière unique un ordinateur sur un réseau. Ces adresses sont attribuées par les administrateurs du réseau ou par une autorité spécifique. Dans notre cas, les machines `dest` et `src` seront à remplacer par les adresses IP de ces machines. Les adresses IP peuvent aussi être associées (de manière unique) à un nom. Ainsi si l'association entre `www.machine.com` et une adresse IP est effectuée, on peut utiliser `www.machine.com` dans la commande `scp`.

4.2 La commande wget

HyperText Transfer Protocol (HTTP))

HTTP est le protocol sous-jacent du World Wide Web. Quand on consulte une page web c'est le protocole HTTP qui est utilisé.

Contrairement à la commande `scp`, la commande `wget` est unidirectionnelle. C'est-à-dire qu'elle ne permet que de copier un fichier se trouvant sur un ordinateur distant vers un l'ordinateur local. La commande `wget` utilise le plus souvent le protocole HTTP bien qu'il puisse communiquer aussi au travers des protocoles comme FTP. La commande `wget` permet donc de copier localement un fichier se trouvant sur un serveur web. L'utilisation la plus courante de la commande `wget` est le téléchargement d'archives conservées sur un serveur web.

```
wget http://www.example.com/archive.tgz
```

La commande précédente permet de télécharger dans le répertoire courant (répertoire depuis lequel la commande `wget` est lancée) l'archive `archive.tgz` se trouvant sur le site web `http://www.example.com/`. À la fin de l'exécution correcte de la commande, une copie du fichier `archive.tgz` se trouvera sur la machine locale. L'option `-c` de la commande `wget` permet la reprise d'un téléchargement.

4.3 La commande `ftp`

File Transfer Protocol (FTP)

FTP est un protocole de transfert de fichier utilisant un modèle client serveur. Il faut donc avoir un client FTP d'un côté et un serveur FTP de l'autre.

La commande `ftp` est une commande permettant de transférer de manière bi-directionnelle des fichiers entre un ordinateur local et un ordinateur distant. La commande `ftp` est une interface utilisateur pour le protocole FTP. La commande `ftp` s'utilise du côté client. La commande `ftp` s'utilise souvent de manière interactive et peut nécessiter ou non une authentification. L'interactivité de la commande `ftp` permet de naviguer dans les répertoires du serveur FTP distant si le serveur l'autorise. L'utilisation principale de la commande `ftp` est le transfert de fichiers volumineux entre le client et le serveur.

Par exemple, la commande `ftp ftp.exemple.com` permet d'ouvrir une session interactive avec le serveur `ftp.exemple.com`. Si le serveur n'autorise pas les connexions anonymes, alors un login et un mot de passe seront demandés. Une fois ce login et mot de passe renseignés, on entre dans une session FTP.

```
$ ls
photo3.png
$ ftp ftp.exemple.com
> ls
archive.tgz text.txt img/
> cd img
> ls
photo1.png photo2.png
> get photo1.png
...
> get photo2.png
...
> put photo3.png
...
> ls
photo1.png photo2.png photo3.png
> bye
$ ls
photo1.png photo2.png photo3.png
```

Dans l'échange précédent, nous avons dans notre répertoire courant un fichier `photo3.png`. Nous nous connectons au serveur ftp `ftp.exemple.com`. Une fois connecté, nous entrons dans la session FTP interactive. La commande `ls` permet de lister les répertoires et les fichiers se trouvant sur le serveur FTP. La commande `cd` permet de changer de répertoire, dans notre cas, nous nous positionnons dans le répertoire `img/`. Dans notre exemple, ce répertoire contient deux fichiers `photo1.png` et `photo2.png`. La commande `get` permet de transférer les fichiers se trouvant sur le serveur vers l'ordinateur local.

Dans notre cas, nous transférons les deux fichiers `photo1.png` et `photo2.png`. La commande `put` permet de transférer un fichier local, dans notre cas `photo3.png`, vers le serveur FTP. Il faut que le fichier `photo3.png` existe localement et se trouve dans le répertoire à partir duquel la commande `ftp` a été lancée. Si le transfert s'est bien passé, nous voyons que le répertoire `img/` de notre serveur FTP contient trois fichiers. La commande `bye` permet de quitter la session FTP interactive et de revenir à son shell. La commande `ls` montre que le répertoire contient maintenant trois fichiers.

Attention, les échanges FTP ne sont en général pas sécurisés. La commande `help` dans une session FTP interactive permet de lister toutes les commandes disponibles.

5 Travailler à distance

Nous avons vu dans la section précédente comment transférer les fichiers. La session interactive FTP nous offre beaucoup de flexibilité mais ne permet pas d'effectuer un vrai travail à distance. Il existe plusieurs commandes permettant d'effectuer du travail à distance, c'est-à-dire d'ouvrir une session interactive sur une machine distante avec un shell complet (en incluant toutes les commandes de Bash par exemple). Historiquement, la commande `telnet` était utilisée pour le travail à distance. Elle n'est cependant plus utilisée à cause de problèmes de sécurité. En effet, la commande `telnet` n'encrypte pas les échanges. La commande utilisée actuellement est la commande `ssh`.

5.1 La commande `ssh`

La commande `ssh` (Secure Shell) permet d'ouvrir une session interactive sur une machine distante mais nécessite d'avoir un nom d'utilisateur et un mot de passe sur cette machine distante. Pour se connecter à la machine `exemple.com` avec le nom d'utilisateur `alice` nous utilisons la commande

```
ssh alice@exemple.com
```

Il faudra renseigner le mot de passe de `alice` pour la machine `exemple.com`. Une fois la connexion sur la machine distante effectuée, on se retrouve avec un shell (Bash ou autre) pour pouvoir effectuer des tâches.

```
bob @ local ~ $ ssh alice@exemple.com
password:
alice @ exemple.com ~ $
```

Dans l'exemple précédent, nous avons changé la variable d'environnement `PS1` pour afficher le nom d'utilisateur, le nom de la machine et le répertoire courant. La variable `PS1` nous indique que l'utilisateur actuel est `bob` sur la machine appelée `local` et qu'il est actuellement dans son répertoire personnel.

Une fois la connexion SSH lancée et le mot de passe renseigné, nous nous retrouvons en tant que `alice` sur la machine `exemple.com` et dans le répertoire de connexion de `alice`.

Il faut noter que si une commande à traitement long est lancée, par exemple `ls -Rl /` et que la connexion s'interrompt, la session interactive est déconnectée et l'exécution de la commande s'arrête. Pour éviter qu'une commande ne s'interrompe en cas de déconnexion, on utilise les commandes `tmux` ou `screen` qui permettent de détacher un terminal sans interrompre la commande. Ainsi, si une déconnexion survient, il suffit de se reconnecter et de re-attacher le terminal (`screen` ou `tmux`) pour voir le résultat de la commande à traitement long.

6 Conclusion

Dans cette activité nous avons vu comment manipuler les fichiers en les compressant ou en les archivant, et en les transférant depuis ou vers un ordinateur distant.

La progression de cette activité s'articule autour du travail à distance qui est l'une des principales utilisations de la ligne de commande. Nous avons vu comment archiver une hiérarchie de fichiers et comment la compresser. Nous avons ensuite vu comment la transférer d'un ordinateur à un autre et enfin, nous avons vu comment nous connecter sur une machine à distance. Nous avons appris comment décompresser et désarchiver des fichiers. Cette procédure dans son intégralité permet de reproduire facilement son environnement de travail sur plusieurs postes distants et ainsi d'être aussi efficace localement qu'à distance.

Conclusion

Vous n'êtes plus un débutant, car vous maîtrisez parfaitement votre environnement de travail. Vous savez manipuler les fichiers et effectuer des transformations complexes sur ces fichiers en une ligne de commande. Cette séquence vous a sûrement permis d'être plus à l'aise devant votre ligne de commande et vous êtes maintenant certainement plus efficace et plus rapide dans son usage. L'adaptation de votre environnement, la manipulation des fichiers, le calcul, le travail à distance couvrent une grande partie de l'utilisation classique des systèmes Unix.

Maintenant que vous n'êtes plus un débutant, il vous reste une dernière étape à franchir. Vous devez devenir un gourou. Pour cela il vous faut continuer à vous accrocher à votre ligne de commande et à découvrir par vous même de nouvelles commandes ou à créer vos propres commandes en combinant les commandes que vous connaissez déjà. Cette création de commande est la dernière étape de votre voyage. Elle sera abordée dans la séquence suivante avec l'écriture de script.

Séquence 4

Automatisez vos travaux

Introduction

Si on résume les épisodes précédents : votre aventure a commencé par un "Premier contact avec le Bash ", où une série d'activités vous ont permis de découvrir le système d'exploitation de type Unix. Vous avez ensuite appris à "Parler son langage" afin d'être en mesure d'interagir efficacement avec ce système. Le Bash est alors devenu votre allié, et dans la 3ième semaine nous avons vu qu'avec lui, vous pouvez maîtriser votre système d'exploitation.

Il est donc temps de franchir l'ultime palier pour être en mesure de "se libérer de la routine" que l'on peut parfois subir quand on est amené à côtoyer intensément un terminal. Et pour cela, nous allons aborder une nouvelle modalité d'usage permise par le Bash : **le mode script**.

Dans le mode script le shell devient un langage de programmation. Et dans les prochaines activités nous allons voir que le shell dispose en effet de tous les ingrédients qui caractérisent un vrai langage informatique : variables, commentaires, branchements conditionnels, structures itératives et la possibilité de définir des fonctions.

Nous allons étudier progressivement chacun de ces éléments dans le cadre de la réalisation de scripts shell, mais gardez bien à l'esprit qu'ils ne sont pas spécifiques au mode script et qu'on peut tout aussi bien les utiliser directement en mode interactif.

Le programme de cette séquence est donc le suivant :
- éléments d'un script shell,
- expressions et conditions,
- structures conditionnelles,
- structures itératives,
- structures de routines.

Activité 4.1

Éléments d'un script shell

1 Introduction

Jusqu'à présent nous avons utilisé le shell en mode interactif, en tapant dans le terminal les lignes de commande successives que le shell doit traiter. Mais on peut également l'utiliser en mode script. Dans ce mode, la suite de lignes de commande est écrite dans un fichier texte qu'on appelle un script. Le nom de ce fichier devient alors une nouvelle commande dont le traitement va consister à déclencher successivement chacune des lignes de commande écrites dans le fichier texte.

Dans cette partie, nous allons voir ce qu'est un script shell à l'aide d'un petit exemple d'illustration. Ensuite nous apprendrons comment utiliser des variables et les arguments des scripts afin de créer des traitements plus sophistiqués.

2 Hello World !

Voyons tout de suite un petit exemple illustratif afin de bien comprendre ce dont il s'agit. En informatique, la tradition veut que le premier programme qu'on écrit quand on apprend un nouveau langage de programmation est le programme : *Hello World*. C'est un programme très simple qui se limite à afficher un message sur l'écran. Nous allons donc voir ici comment s'écrit un tel programme dans un script shell.

Comme nous l'avons dit, un script est un simple fichier texte qui contient des commandes shell. Nous allons donc commencer par créer un tel fichier. Vous pouvez utiliser n'importe quel éditeur de texte. Par exemple l'éditeur `vi` [1] conviendra très bien. On va choisir de créer un nouveau fichier sous le nom `hello.sh` :

```
$ vi hello.sh
```

L'extension ".sh" n'est pas obligatoire, mais nous prendrons l'habitude de l'utiliser afin de pouvoir reconnaître facilement nos scripts. La première ligne que nous allons écrire dans ce fichier sera toujours la même :

```
#!/bin/bash
```

On appelle cette ligne le "*shebang*". Ce nom vient de la contraction de *sharp* et de *bang*, en référence aux deux premiers caractères de cette ligne : le croisillon '#' (*sharp* en anglais), et le point d'exclamation

1. Souvenez vous qu'avec `vi` il faut commencer par appuyer sur la touche [i] pour passer en mode d'édition et sortir de ce mode par la touche [Esc] et de l'éditeur par `:wq`

'!' (*bang*). Le *shebang* doit toujours être écrit au tout début du fichier script. Il permet d'indiquer au système l'interpréteur à utiliser pour les lignes qui suivent dans le reste du fichier. Dans notre cas, nous indiquons que le script doit être traité par le Bash.

Nous pouvons ensuite sauter une ou plusieurs lignes. Cela n'a pas d'incidence, mais ça rend le contenu plus lisible. Pour ce premier exemple, nous allons faire un script minimaliste qui ne va contenir que deux commandes d'affichage.

```
#!/bin/bash

echo "Hello"
echo  "world !!!"
```

Voilà, on enregistre le fichier et on quitte l'éditeur de texte. À ce stade, il nous faut faire encore une dernière manipulation. Dans la mesure où nous allons devoir lancer l'exécution de ce script, il nous faut au préalable nous assurer que ce fichier dispose bien du droit "exécuter". Nous devons donc taper la commande :

```
$ chmod +x hello.sh
```

À partir de maintenant, nous n'aurons plus besoin de re-spécifier ce droit pour ce fichier, même si nous modifions son contenu à l'avenir. Aussi nous n'aurons plus besoin d'utiliser cette commande.
Tout est donc en place. Nous pouvons maintenant tester notre script en lançant son exécution. Pour ce faire, il nous suffit de taper le nom du fichier dans la console, comme on le ferait pour une commande classique.

```
$ ./hello.sh
Hello
world !!!
$
```

Notez que nous avons fait précéder le nom du script par le chemin d'accès au script avec les symboles `./`, car le fichier script se trouve dans le répertoire courant. Si le script se trouve dans un autre répertoire, il faudra indiquer le chemin menant au fichier script pour que cela fonctionne. Il est possible de se passer de spécifier le chemin d'accès au répertoire courant, si le caractère '.' (point) se trouve dans la variable d'environnement `PATH`. Aussi, si vous souhaitez bénéficier de cet allègement d'écriture, vérifiez la présence du point dans la variable `PATH` et ajoutez-le s'il n'est pas présent.

✍ **Challenge C41Q1**

Alice fait un loto

3 Les variables dans un script

Un vrai script va généralement réaliser un traitement un peu plus complexe qu'un simple affichage "Hello World". Et nous aurons souvent besoin de manipuler des variables durant ce traitement. La syntaxe de manipulation des variables est la même que celle du mode interactif. Et pour rappel, les variables shell ont les particularités suivantes :

- elles ne sont pas typées,
- elles n'ont pas besoin d'être déclarées,
- leur contenu par défaut est vide, c'est-à-dire la chaîne vide,
- pour accéder au contenu d'une variable, il faut préfixer son nom par le caractère `'$'`.

La différence avec le mode interactif est qu'ici les variables ne vont exister que pendant l'exécution du script. Elles seront en quelque sorte locales au script. Voyons cela en pratique. Nous allons éditer un nouveau fichier de nom `henri.sh` :

```
$ vi henri.sh
```

Dans ce script on va utiliser deux variables `age` et `nom`, et leur affecter les valeurs 45 et `Henri`.

```
#!/bin/bash
age=45
nom=Henri
```

Ici les règles syntaxiques sont exactement les mêmes que celles du mode interactif. L'affectation d'une variable doit se faire sans espace autour du caractère `'='`. La valeur à affecter peut être une chaîne de caractères ou un nombre. Si la chaîne contient des espaces, il faut supprimer la fonction de séparateur de mots de l'espace en utilisant une inhibition (voir l'activité 2.3). Cette valeur peut aussi être obtenue par une des différentes constructions syntaxiques que vous avez étudiées dans les activités précédentes. Citons en particulier :

- pour le contenu d'une variable, la substitution de variable : `$variable`
- pour le code retour de la dernière commande exécutée : `$?`
- pour le résultat d'un calcul, la substitution arithmétique : `$((calcul))`
- pour le résultat d'une commande, la substitution de commande : `$(commande)`

Maintenant, dans notre exemple, nous allons exploiter les deux variables dans une commande d'affichage `echo` :

```
#!/bin/bash
age=45
nom=Henri
echo "Monsieur $nom a $age ans"
```

Comme vous le savez, le symbole `'$'` va conduire à remplacer le nom de la variable par son contenu. La substitution de variable opère dans une inhibition partielle, à savoir entre deux guillemets. Une fois l'édition terminée, on enregistre, on quitte et on fixe le droit "exécutable" sur le fichier script :

```
$ chmod +x  henri.sh
```

Maintenant si on lance l'exécution du script, on voit le texte s'afficher.

```
$ ./henri.sh
Monsieur Henri a 45 ans
$
```

On peut constater qu'après l'exécution, les variables `age` et `nom` n'existent pas dans le shell :

```
$ ./henri.sh
Monsieur Henri a 45 ans
$ echo $age
      (<- pas d'affichage)
$ echo $nom
      (<- pas d'affichage)
$
```

Cela s'explique par le fait que le script a été exécuté par un processus fils au shell. Nous reviendrons sur ce phénomène de localité un peu plus loin.
Pour être complet, on peut rappeler que la substitution de variable s'écrit dans sa forme complète

avec des accolades : `${variable}`. La forme complète s'utilise quand le caractère qui suit le nom de la variable est un caractère alphanumérique ou le caractère souligné. Ainsi, si dans notre script `henri.sh` nous devons faire un pseudonyme composé du nom et de l'âge, séparés par la lettre `A`, nous allons avoir besoin d'utiliser la forme complète. Voyons ce que cela donne :

```
#!/bin/bash
age=45
nom=Henri
pseudo1="$nomA$age"
echo $pseudo1             # -> 45
pseudo2="${nom}A$age"
echo $pseudo2             # -> HenriA45
```

Pour l'affectation de `pseudo1`, le nom de la première variable pose problème car il va être interprété comme le nom `nomA` au lieu du nom de variable `nom` suivi de la lettre `A`. L'affichage va ainsi donner `45` car `nomA` a la valeur vide. En ayant recours aux accolades comme pour `pseudo2`, le nom de la variable est isolé. Et dans ce cas, le bon résultat s'affiche : `HenriA45`.

```
$ ./henri.sh
45
HenriA45
$
```

Cet exemple montre aussi comment ajouter des commentaires dans le script. Il suffit d'utiliser le symbole '`#`', tout ce qui va être écrit après jusqu'au retour à la ligne sera ignoré par le shell. Ce qui est parfait pour y placer des remarques et des notes explicatives.

Exercice 4.1.1: Écrire le code d'un script de nom `sumvar.sh`, ce script doit commencer par définir deux variables contenant chacune une valeur numérique, puis afficher le résultat de la somme de ces deux variables.

Solution page 211

3.1 La portée des variables

Dans un script, on dit que les variables sont locales car les affectations de variables qu'on réalise ne vont pas se répercuter dans le contexte du processus qui a lancé l'exécution du script (i.e. le shell de votre terminal). Par exemple, si on exécute la dernière version du script `henri.sh`, puis si nous essayons d'afficher le contenu de la variable `pseudo2` depuis le shell du terminal, rien ne va s'afficher :

```
$ ./henri.sh
45
HenriA45
$ echo $pseudo2

$
```

En effet, les commandes shell sont traitées dans un processus auquel est rattaché un ensemble de variables : on appelle cet ensemble un *contexte d'exécution*. Les commandes d'un script sont traitées dans un contexte d'exécution différent de celui du shell du terminal sur lequel il a été lancé. Et à la fin de l'exécution du script, son contexte d'exécution est supprimé. Ainsi, au niveau du terminal, les opérations d'affectation réalisées par le script sont donc sans effet.

3.2 "Sourcer" un script

Il existe un moyen de contourner ce comportement de localité des variables manipulées au sein d'un script. Il faut pour cela lancer l'exécution du script dans le même contexte d'exécution que le shell du terminal. Ceci se fait à l'aide de la commande `source` qui se note de manière abrégée par le symbole point ('.') devant le nom du script à exécuter. On dit alors qu'on "*source*" le script.

```
$ source ./henri.sh # ou . ./henri.sh
45
HenriA45
$ echo $pseudo1
45
$
```

Si vous consultez les PID, par exemple avec la commande `ps`, vous remarquerez que le PID n'a pas changé lorsque le script est sourcé. Ce qui n'est pas le cas quand le script est lancé comme une commande. Quand il n'est pas sourcé, le lancement du script crée un nouveau processus qui va se charger de réaliser l'exécution du code du script. Ce nouveau processus est ce que l'on a appelé un sous-shell dans les séquences précédentes.

3.3 Utiliser des variables externes

On peut aussi vouloir faire l'inverse. C'est-à-dire utiliser dans le script des variables qui sont fixées avant de lancer son exécution. Et là aussi il nous faut être vigilant car le contexte d'exécution du shell du terminal ne se transmet pas systématiquement au nouveau processus qui va être créé pour exécuter le script. Mettons cela en évidence en commentant l'affectation de la variable `AGE` dans notre script `henri.sh` :

```
#!/bin/bash
#AGE=45
NOM=Henri
[...]
```

Si on affecte 30 à cette variable dans le shell du terminal, avant de lancer le script, cela n'aura aucun effet. L'affichage produit par le script ne va pas faire apparaître l'âge :

```
$ AGE=30
$ ./henri.sh

HenriA
$
```

Cela est une nouvelle fois dû au fait que le shell du terminal et le script sont exécutés dans deux processus différents et donc dans deux contextes d'exécution différents. On peut bien sûr s'en sortir en "sourçant" le script, mais s'il s'agit de ne transmettre qu'une seule variable il est préférable d'utiliser la commande `export` (comme vu à l'activité 3.1). Cette commande effectuée dans le terminal du shell indique que la variable en argument devra être copiée dans tous les contextes d'exécution des processus qui seront créés par la suite. Essayons cela sur la variable `AGE` :

```
$ export AGE=30
$ ./henri.sh
30
HenriA30
$
```

On a exporté cette variable en lui affectant 30, juste avant de lancer le script `henri.sh`. Et on remarque cette fois que le script a bien pris en compte le contenu de la variable du processus parent. Attention, cela ne veut pas dire que les modifications faites sur une variable exportée dans le processus d'un script seront répercutées au niveau du shell du terminal.

3.4 Résumé sur la portée des variables

Pour résumer, concernant la portée des variables dans un script, nous avons les trois situations suivantes :

- Le cas le plus courant où on lance simplement le script. Ce dernier dispose alors d'un contexte d'exécution qui lui est propre. Toutes ses variables sont locales.
- Le cas où on source un script. Le script partage alors le même contexte d'exécution que son shell de lancement. Toutes les variables sont partagées.
- Enfin, le cas où un export de une ou plusieurs variables a été fait avant de lancer le script. Les variables exportées sont copiées dans le contexte d'exécution qui sera créé lors du lancement du script. Seules ces variables conserveront leur contenu dans le script.

4 Les arguments d'un script : Les variables positionnelles

Comme pour n'importe quelle commande, des arguments peuvent être passés à un script par la ligne de commande. Dans le code du script, ces arguments sont placés dans des variables particulières qui sont automatiquement renseignées au début de l'exécution du script. On qualifie ces variables de positionnelles. De plus, les variables positionnelles peuvent être renseignées lors d'un appel d'une fonction (comme vous le verrez dans l'activité 4.5) ou lors de la commande interne `set`). Les variables positionnelles ont la particularité d'être nommées par un numéro.

Argument et paramètre

Les arguments d'un script sont placés dans des variables positionnelles. Ces variables sont nommées plus généralement des paramètres. Un argument représente la valeur donnée au lancement d'un script. Le paramètre du script est la variable positionnelle qui contient l'argument et utilisée par les commandes dans le script

La syntaxe de consultation est `$`*`n`* où *n* indique la position de l'argument à l'appel. Pour les variables positionnelles ayant plus de un chiffre, il faut avoir recours aux accolades `${`*`nn`*`}`.

La commande interne `set arg1 ...` affecte ses arguments aux paramètres positionnelles en commençant au numéro 1. Illustrons, l'affectation par la commande `set` et l'accès aux variables :

```
$ set 1 2 3 4 5 6 7 8 9 a b
$ echo $1 $9 ${10}
1 9 a
```

En plus des variables positionnelles, des variables spéciales sont chargées pour décrire la liste des arguments. Ces variables spéciales sont les suivantes :

- `$0` : le nom du script,
- `$#` : le nombre d'arguments existants,
- `$*` : la liste de tous les arguments,
- `$@` : la liste de tous les arguments.

La différence entre `$*` et `$@` porte sur la méthode d'affectation des arguments de la ligne de commande. La variable `$@` va servir à prendre en compte les espaces au sein d'un argument lorsque la variable est utilisée entre guillemets de cette manière `"$@"`.

Pour mettre ces éléments en pratique, nous allons créer un nouveau script sous le nom `nbArg.sh`. Ce script va contenir le code suivant :

```
#!/bin/bash
echo "Il y a $# argument(s)"
echo  "Le 1er  est : $1"
```

Ainsi, ce script va afficher le nombre d'arguments qu'il reçoit lors de son appel, puis le contenu du premier d'entre eux. Si on le lance sans argument, il va afficher 0 comme nombre d'arguments, et rien pour la valeur du premier argument :

```
$ chmod +x  nbArg.sh
$ ./nbArg.sh
Il y a 0 argument(s)
Le 1er est :
$
```

Et naturellement, si on passe des arguments, l'affichage produit sera cohérent :

```
$ ./nbArg.sh  yop yep yap
Il y a 3 argument(s)
Le 1er est : yop
$
```

Exercice 4.1.2: Écrire le code d'un script de nom `swap.sh` qui doit afficher les deux premiers arguments qu'il reçoit en inversant leur position. Exemple d'appel :

```
$ swap.sh Alice Bob
Bob Alice
$
```

Solution page 211

4.1 Décalage des variables positionnelles

La commande `shift` décale vers la "gauche" les variables positionnelles. Par exemple, si dans un script la variable `$1` vaut 'a' et `$2` vaut 'b', la commande `shift` va décaler le contenu de la variable positionnelle `$2` sur la variable positionnelle `$1`. Le contenu de `$1` qui était préalablement à la commande `shift` est perdu. Le nombre de variables positionnelles est décrémenté de un.

Pour illustrer cette commande, prenons le script `decal.sh` avec le code suivant :

```
#!/bin/bash
echo $# "$@"
shift
echo $# "$@"
```

L'exécution de ce script montre que le contenu du premier argument est perdu et qu'il n'y a plus que deux variables positionnelles.

```
$ decal.sh  a b c
3 a b c
2 b c
$
```

4.2 Affectation des variables positionnelles

Avant d'aller plus loin, revenons sur la commande interne `set`. Depuis l'activité 3.1, nous savons que la commande `set` exécutée sans option et sans argument affiche le nom et la valeur de chaque variable du shell.

Si on passe des arguments à la commande `set`, ceux-ci sont affectés aux variables positionnelles. Plus précisément, les variables positionnelles sont re-initialisées avec les nouveaux arguments. L'exemple ci-dessous montre que les variables positionnelles initialisées par la première commande `set` ont été supprimées par la seconde commande `set`.

```
$ set  a b c
$ echo $#
3
$ set a
$ echo $#
1
```

La commande `set` accepte aussi des options. Pour éviter que le premier argument, s'il commence par un tiret, ne soit considéré comme une option et entraîne une erreur dans l'interprétation des arguments, on écrira les arguments après le double tiret `'--'`. Ce double tiret signifie en Bash la fin des options. Ce qui suit le double tiret est donc obligatoirement un argument même si l'argument commence par un tiret. La commande `set --` sans argument supprime toutes les variables positionnelles.

```
$ set -- -a b
$ echo $@ $#
 -a b 2
$ set --
$ echo $#
0
```

Classiquement les arguments d'une commande sont des mots séparés par l'espace. Le séparateur de mots est défini dans la variable d'environnement IFS (*Internal Field Separator*). En modifiant la valeur de l'IFS, la commande `set` peut servir à découper une chaîne de caractères en plusieurs mots et à les associer aux variables positionnelles.

```
$ ligne=alice:Liddell
$ IFS=':'
$ set -- $ligne
$ echo $1 $2
alice Liddell
```

4.3 Prudence vis-à-vis du contenu des variables

Quand des variables sont placées comme arguments d'un script ou comme arguments d'une commande, il y a une précaution à prendre pour éviter des erreurs lors de l'exécution du script.
Par exemple, dans l'écriture `echo $var`, le contenu de la variable `var` est passé comme argument de la commande `echo`. Dans cette situation, il est conseillé d'effectuer la substitution de variable dans une inhibition partielle (des guillemets) de la manière suivante `echo "$var"`. Car si la variable utilisée contient une chaîne de caractères comprenant des espaces, en l'absence de l'inhibition partielle, les espaces vont être traitées par le shell comme séparateur de mots et donc à l'appel de la commande chaque mot sera vu comme un argument supplémentaire, au lieu d'en former un seul à eux tous. Pour vous en convaincre, faisons l'expérience suivante :

```
$ TEXTE="Bonjour tous"
$ echo $TEXTE
Bonjour tous
$
```

A priori, le contenu de la variable **TEXTE** ne pose ici aucun problème. Mais si on utilise notre script **nbArg.sh** comme commande à la place de **echo**, regardons ce que cela produit :

```
$ TEXTE="Bonjour tous"
$ ./nbArg.sh  $TEXTE     #le script nbArg
Il y a 2 argument(s)
Le 1er  est : Bonjour
$
```

Nous constatons que le script affiche seulement le mot "Bonjour" et pas l'intégralité du contenu de la variable. L'espace a en effet séparé la chaîne de caractères en deux arguments. Pour inhiber les espaces, il faut utiliser l'inhibition partielle. C'est-à-dire encadrer la substitution de variable par des guillemets. Et cette fois l'affichage sera bien cohérent :

```
$ TEXTE="Bonjour tous"
$ ./nbArg.sh  "$TEXTE"     #le script nbArg
Il y a 1 argument(s)
Le 1er  est : Bonjour tous
$
```

Aussi, comme on ne sait pas par avance si le contenu d'une variable contient ou non des espaces, nous prendrons l'habitude de toujours encadrer la substitution de variable par des guillemets pour éviter toute mauvaise surprise.

4.4 Manipulation avancée des arguments

Pour mieux exploiter les arguments d'un script il est courant d'utiliser les syntaxes avancées de la substitution de variable. Cette syntaxe est utilisée principalement pour des scripts interactifs dans lesquels il est prudent d'avoir un contrôle sur la saisie de l'utilisateur.

Par exemple, pour éviter qu'une variable positionnelle ne retourne pas une valeur vide si elle ne correspond à aucun argument ; on peut spécifier une substitution de variable avec valeur par défaut. La valeur par défaut est indiquée après le symbole `':-'` et se note de la manière suivante :

```
${nom_variable:-defaut}
```

La valeur par défaut n'est pas affectée à la variable, elle est juste retournée à l'accès au contenu de la variable.

Pour contrôler la présence d'un contenu associé à une variable, il y a la syntaxe :

```
${nom_variable:?message}
```

Cette substitution de variable indique que la valeur de la variable doit être retournée si elle existe, ou sinon que le message d'erreur indiqué après le symbole `?` est affiché et l'exécution du script est interrompue. Cette syntaxe trouve son utilité pour vérifier la présence d'un argument obligatoire.

Nous allons respectivement voir leur usage dans deux petits scripts. Nous allons commencer par une amélioration de notre script **nbArg.sh**. On souhaite cette fois afficher le mot "vide" si le premier

argument n'est pas présent. On va donc utiliser la syntaxe qui permet d'affecter une valeur par défaut en cas d'absence du premier argument.

```
#!/bin/bash
echo "Il y a $# argument(s)"
FIRST=${1:-vide}
echo  "Le 1er  est : $FIRST"
```

Si on lance l'exécution du script, l'affichage produit est maintenant plus parlant :

```
$ ./nbArg.sh
Il y a 0 argument(s)
Le 1er est : vide
$
```

Pour illustrer la deuxième syntaxe, nous allons créer un nouveau script `arg1.sh`, qui exige au moins un argument à son lancement. Ici il nous faut utiliser la syntaxe qui va contrôler l'existence du premier argument. Dans celle-ci on va renseigner la phrase d'erreur à afficher en cas d'absence de cet argument.

```
#!/bin/bash
ARG1=${1:?"Vous devez fournir un argument"}
echo  "Le 1er  est : $ARG1"
```

Si on lance ce script sans argument, ce message d'erreur va s'afficher :

```
$ chmod +x  arg1.sh
$ ./arg1.sh
./arg1.sh  line 3:  1: Vous devez fournir un argument
```

Notez que l'exécution s'est brutalement stoppée : la commande `echo` n'a pas eu le temps d'être déclenchée.

✍ **Challenge C41Q2**

Alice fait des mathématiques

5 Le code retour d'un script

Quand une commande termine son exécution, elle renvoie un code retour (voir l'activité 2.4) pour indiquer au processus parent (celui qui a créé le processus de la commande), si l'exécution s'est bien passée (avec la valeur 0), ou si une erreur s'est produite (avec une valeur entre 1 et 255). C'est la même chose pour un script. Le code retour retourné par un script est :

- Soit celui spécifié par la commande : `exit` *`code`*, qui va mettre explicitement fin à l'exécution du script en renvoyant le code retour *`code`*. L'argument *`code`* est optionnel, et vaut 0 par défaut.
- Soit celui renvoyé par la dernière commande exécutée dans le code du script.

Le code retour se récupère par l'appelant du script en utilisant `$?`.

Voici un petit exemple de consultation de ce code retour avec le script `arg1.sh` :

```
$ ./arg1.sh  coucou     #le script arg1
Le 1er  est : coucou
$ echo $?
```

```
0
$
```

Comme on passe un argument à ce script, quand on consulte ensuite la valeur `$?` on obtient la valeur 0. Car la valeur 0 est en fait la valeur retournée par la commande `echo` située à la fin du script `arg1.sh`. Ce 0 signifie donc que la commande `echo` s'est déroulée sans erreur.
Mais si on relance le script sans mettre d'argument, nous aurons cette fois la valeur `1` dans `$?` :

```
$ ./arg1.sh
./arg1.sh: line 3: 1: Vous devez fournir un argument
$ echo $?
1
```

Ici la valeur de `$?` est celle retournée par la dernière commande traitée dans `arg1.sh`, qui est cette fois la commande d'affectation de la variable `ARG1`. Dans cette commande, la syntaxe utilisée pour vérifier la présence de l'argument `1` conduit à produire une erreur :

```
ARG1=${1:?"Vous devez fournir un argument"}
```

Cela a donc stoppé l'exécution du script, et ce dernier renvoie le code retour de cette dernière commande, soit `1` signifiant la présence d'une erreur.

6 Débogage d'un script

Il est possible d'afficher chacune des commandes exécutées par un script à l'aide de l'option d'exécution `-x` du Bash :

```
$ bash -x script_a_lancer
```

Chaque commande exécutée sera alors affichée dans la console sur une ligne qui commencera par le caractère `'+'`. On peut aussi demander l'affichage des lignes et des blocs lus avant leur traitement en ajoutant l'option `-v` :

```
$ bash -x -v script
```

Enfin, avant d'exécuter un script, il peut déjà être intéressant de savoir si la syntaxe du Bash a été respectée. Dans ce cas, l'option `-n` va vous permettre de lever les erreurs de syntaxe ou de vérifier l'absence d'erreur.

```
$ bash -n script
```

On peut aussi effectuer les opérations de débogage sur une partie du script. La section du script à étudier est encadrée par la commande `set`. Le début de la section est marquée avec la commande `set -x`. La fin de la section est indiquée par `set +x`.

7 Conclusion

Dans cette activité nous avons vu comment :
- écrire un script,
- lancer son exécution,
- utiliser des variables dans un script,

— exploiter les arguments fournis dans la ligne de commande,
— retourner un code retour depuis un script,
— et comment tracer l'exécution des commandes contenues dans un script.

Vous savez donc maintenant écrire des scripts shell !

✍ **Challenge C41Q3**

Tu cut, tu tag, tu move !

Solutions des exercices

Solution de l'exercice 4.1.1 page 202:

```
#!/bin/bash
val1=56
val2=67
echo $((val1 + val2))
```

Solution de l'exercice 4.1.2 page 205:

```
#!/bin/bash
echo "$2  $1"
```

Activité 4.2

Expressions et conditions

1 Introduction

Comme tout langage de programmation, le Bash fournit des structures de contrôle permettant de prendre des décisions lors de l'exécution d'un script. Ces décisions sont prises à l'aide de structures de contrôle et de la réalisation de tests.

Toutes les commandes d'un système Unix ont un code retour égal à 0 si la commande s'est terminée normalement et entre 1 et 255 sinon. En shell Bash on peut donc voir toute commande comme une expression conditionnelle dont le code retour est assimilé à vrai s'il vaut 0 et à faux dans les autres cas.

Dans ce chapitre, nous allons voir comment utiliser ce code retour pour effectuer des tests sur des nombres, sur des chaînes de caractères et sur des fichiers.

2 La commande `test`

Supposons qu'on veuille tester si la valeur d'une variable numérique `a` est plus grande que 10. Avec la représentation mathématique usuelle, cette condition s'écrit $a > 10$. Le shell propose deux écritures pour effectuer ce test :

— soit on écrit la commande `test` suivie de l'expression à tester :
```
test $a -gt 10
```
— soit on écrit l'expression à tester entre crochets :
```
[ $a -gt 10 ]
```
Attention à cette écriture : l'espace suivant le crochet ouvrant et l'espace précédant le crochet fermant sont obligatoires ! Il en est de même des espaces autour des opérateurs utilisés dans l'expression à tester.

Ces deux écritures sont équivalentes, elles permettent de tester la valeur d'une expression conditionnelle et renvoie comme code retour 0 si le test est vrai et 1 si le test est faux.

2.1 Opérateurs de comparaison arithmétique

La commande `test` dispose des principaux opérateurs de comparaison arithmétique sur les entiers. Ils sont introduits sous la forme d'options de commande, c'est-à-dire par un tiret, et suivies de deux lettres qui sont des abréviations de leurs noms en anglais.

Voici la liste de ces opérateurs ; ils s'utilisent avec des nombres entiers ou des variables numériques entières :

`test` *n1* `-eq` *n2*
: vrai si le nombre *n1* est égal au nombre *n2*

`test` *n1* `-ne` *n2*
: vrai si le nombre *n1* est différent du nombre *n2*

`test` *n1* `-gt` *n2*
: vrai si le nombre *n1* est strictement supérieur (*greater than*) au nombre *n2*

`test` *n1* `-lt` *n2*
: vrai si le nombre *n1* est strictement inférieur (*less than*) au nombre *n2*

`test` *n1* `-ge` *n2*
: vrai si le nombre *n1* est supérieur ou égal (*greater or equal*) au nombre *n2*

`test` *n1* `-le` *n2*
: vrai si le nombre *n1* est inférieur ou égal (*less or equal*) au nombre *n2*

Dans les exemples qui suivent et qui sont réalisés en ligne de commande, on visualise le résultat de la commande `test` en affichant son code retour qui est contenu dans la variable `$?`

Répétons-le encore une fois : un code retour 1 signifie que le test est faux, 0 que le test est vrai.

```
$ a=3
$ test $a -gt 12; echo $?
1
$ test $a -lt 0; echo $?
1
$ test $a -ge 2; echo $?
0
$ [ $a -ne 3 ]; echo $?
1
$ [ 3 -eq $a ]; echo $?
0
```

2.2 Opérateurs sur les chaînes de caractères

Pour effectuer des comparaisons sur des chaînes de caractères, nous disposons des opérateurs suivants :

`test -n` *chaine*
: vrai si *chaine* est une chaîne de caractères non vide

`test -z` *chaine*
: vrai si *chaine* est une chaîne de caractères vide

`test` *chaine1* `=` *chaine2*
: vrai si *chaine1* est identique à *chaine2*

`test` *chaine1* `==` *chaine2*
: vrai si *chaine1* est identique à *chaine2*

`test` *chaine1* `!=` *chaine2*
: vrai si *chaine1* est différente de *chaine2*

`test` *chaine1* `\<` *chaine2*
: vrai si *chaine1* est inférieure à *chaine2*

`test` *chaine1* `\>` *chaine2*
vrai si *chaine1* est supérieure à *chaine2*

Attention < et > sont des caractères spéciaux du shell. Souvenez-vous de l'activité 2.5 : ces caractères redirigent l'entrée et la sortie standard d'une commande. Pour ne pas avoir de surprise, il faut donc bloquer leur comportement par défaut en les précédant du caractère d'inhibition de caractère (anti-slash \).

À propos de la comparaison de chaînes de caractères

La comparaison de deux chaînes de caractères s'effectue d'une manière analogue à la comparaison de deux mots dans un dictionnaire : on peut dire qu'un mot est inférieur à un autre s'il arrive avant dans l'ordre alphabétique du dictionnaire.

Comme les chaînes de caractères ne sont pas constituées uniquement de lettres mais de n'importe quels caractères, la comparaison s'effectue suivant l'ordre dans lequel les caractères apparaissent dans le codage ASCII (*American Standard Code for Information Interchange*).

Pour comprendre comment sont comparées les chaînes, retenez que dans le code ASCII :
- les lettres de l'alphabet sont rangées dans l'ordre habituel ;
- les lettres majuscules et minuscules sont différenciées et les lettre majuscules (codes 65 à 90) sont rangées avant les lettres minuscules (code 97 à 122), elles sont donc « inférieures » aux lettres minuscules ;
- les caractères représentant les chiffres (codes 48 à 57) sont aussi rangés dans l'ordre et avant les lettres, ils sont donc « inférieurs » aux lettres.

Voici quelques exemples pour illustrer la comparaison des chaînes de caractères :

```
$ mot1=alice
$ echo $mot1
alice
$ [ -n $mot1 ]; echo $?
0
```

Le code retour 0 indique que le test est vrai. La chaîne contenue dans la variable `mot1` n'est effectivement pas vide.

```
$ [ $mot1 = ALICE ]; echo $?
1
```

Le code retour 1 indique que le test est faux. La chaîne contenue dans la variable `mot1`, c'est à dire la chaîne `alice`, n'est pas égale à la chaîne `ALICE`. En effet, les lettres minuscules et majuscules sont distinctes.

```
$ [ $mot1 \> Alice ]; echo $?
0
```

Le code retour 0 indique que le test est vrai, c'est à dire que la chaîne `alice` est supérieure à la chaîne `Alice`. En effet, dans le codage ASCII le caractère `a` vient après le caractère `A` ; autrement dit `a` est supérieur à `A`.

```
$ [ bob \> alice ]; echo $?
0
```

Le code retour 0 indique que la chaîne `bob` est supérieure à la chaîne `alice`. On voit dans cet exemple que ce n'est pas le nombre de caractères qui compte, c'est l'ordre ASCII des caractères, et comme le caractère `b` vient après le caractère `a`, la chaîne `bob` est bien supérieure à la chaîne `alice`.

Attention à ne pas confondre chaîne de caractères et nombre.

Les opérateurs `<` et `>` peuvent être source d'erreurs si vous les utilisez avec des nombres car ce ne sont pas les valeurs numériques qui seront comparées mais des chaînes de caractères constituées de caractères chiffres.

Par exemple, le test suivant dont le résultat est vrai est trompeur car il ne compare pas les valeurs arithmétiques 32 et 20 mais les chaînes de caractères `"32"` et `"20"` et comme le caractère `3` vient bien après le caractère `2` dans le codage ASCII, la réponse est vrai.

```
$ [ 32 \> 20 ]; echo $?
0
```

On se rend mieux compte du problème avec le test suivant qui peut paraître contre-intuitif.

```
$ [ 32 \> 200 ]; echo $?
0
```

Le code retour 0 indique que le test est vrai. En effet, ce sont les chaînes de caractères `"32"` et `"200"` qui sont comparées lorsqu'on utilise l'opérateur `>`, non les nombres 32 et 200. Et comme le caractère `3` vient après le caractère `2` dans le codage ASCII, la chaîne `"32"` est effectivement supérieure à la chaîne `"200"`. Pour effectuer le test sur les nombres 32 et 200 il faut utiliser l'opérateur de comparaison arithmétique `-gt` vu plus haut.

Exercice 4.2.1: Écrire un script shell de nom `aumoinsdeux.sh` dont le code retour est 0 si on lui donne au moins deux arguments et 1 sinon. N'oubliez pas le *shebang* et pensez à attribuer le droit exécutable à votre script afin de pouvoir le tester. Après chaque exécution de votre script, affichez le code retour `$?` pour vérifier s'il a le comportement attendu.
Exemples d'exécution :

```
$ ./aumoinsdeux.sh
$ echo $?
1
$ ./aumoinsdeux.sh aaa; echo $?
1
$ ./aumoinsdeux.sh aaa bbb; echo $?
0
$ ./aumoinsdeux.sh aaa bbb ccc; echo $?
0
```

Solution page 224

2.3 Opérateurs sur les fichiers

La commande **test** fournit beaucoup d'opérateurs pour réaliser des tests sur les fichiers. En voici quelques-uns parmi les plus couramment utilisés ; pour en avoir la liste exhaustive, le lecteur est encouragé à utiliser la commande **man**.

`test -s` *fichier*
: vrai si *fichier* n'est pas vide

`test -f` *fichier*
: vrai si *fichier* existe et est un fichier ordinaire

`test -d` *fichier*
: vrai si *fichier* existe et est un répertoire

`test -e` *fichier*
: vrai si *fichier* existe, quel que soit son type

`test -r` *fichier*
: vrai si *fichier* existe et est accessible en lecture

`test -w` *fichier*
: vrai si *fichier* existe et est accessible en écriture

`test -x` *fichier*
: vrai si *fichier* existe et possède le droit exécutable

`test` *fichier1* `-nt` *fichier2*
: vrai si *fichier1* et *fichier2* existent et que *fichier1* est plus récent que *fichier2*

`test` *fichier1* `-ot` *fichier2*
: vrai si *fichier1* et *fichier2* existent et que *fichier1* est plus ancien que *fichier2*

Illustrons ces opérateurs par quelques exemples :

```
$ > monfichier
$ [ -s monfichier ]; echo $?
1
```

Juste après avoir créé ou écrasé un fichier **monfichier** avec la commande `> monfichier`, le code retour 1, c'est à dire faux, du test effectué indique que le fichier est vide (**-s** est vrai si le fichier est non vide), ce qu'on peut vérifier en listant les propriétés de **monfichier**.

```
$ ls -l monfichier
-rw-r--r--  1 alice  user  0 22 sep 11:25 monfichier
$
```

Ajoutons maintenant du contenu dans **monfichier** et refaisons le test.

```
$ echo Bonjour le monde >> monfichier
$ [ -s monfichier ]; echo $?
0
```

Une fois qu'on a écrit dans **monfichier**, il n'est évidement plus vide !

Créons maintenant un répertoire **monrep** afin de tester les types de fichiers.

```
$ mkdir monrep
$ [ -f monrep ]; echo $?
1
$ [ -f monfichier ]; echo $?
```

```
0
$ [ -d monrep ]; echo $?
0
```

On a vérifié que `monrep` n'est pas un fichier ordinaire, contrairement à `monfichier` et que c'est bien un répertoire.

```
$ [ -w monfichier ]; echo $?
0
```

Si on a pu écrire dans `monfichier`, c'est parce que celui-ci est accessible en écriture, ce que le test ci-dessus permet de vérifier.

```
$ [ monrep -nt monfichier ]; echo $?
0
```

Le répertoire `monrep` a été créé après le fichier `monfichier`, il est effectivement plus récent.

```
$ [ -e zorglub ]; echo $?
1
```

Enfin, ce dernier test indique que le fichier `zorglub` n'existe pas.

2.4 Opérateurs logiques

Naturellement il est possible de combiner les tests vus dans les sections précédentes à l'aide d'opérateurs logiques :

`!` est l'opérateur logique de négation, le NON
`-a` est l'opérateur logique binaire ET
`-o` est l'opérateur logique binaire OU

Ces opérateurs respectent les règles de priorité habituelles de la logique :
`!` est prioritaire sur `-a` et `-o`
`-a` est prioritaire sur `-o`

Voyons à travers quelques exemples l'utilisation des opérateurs logiques pour réaliser des tests complexes.

Imaginons une situation où nous devons vérifier que si un fichier de nom `confidentiel` existe, alors celui-ci ne doit pas être accessible en lecture.

Avec l'opérateur de négation nous pouvons vérifier que le fichier n'est pas accessible en lecture :

```
$ [ ! -r confidentiel ]; echo $?
0
```

Le code retour 0 signifie que le test est vrai, c'est-à-dire que le fichier n'est pas accessible en lecture. Cependant ce test ne permet pas de savoir si le fichier existe. En effet, s'il n'existe pas, il n'est pas accessible en lecture.

Essayons de compléter le test avec l'opérateur logique `-a` pour vérifier que le fichier existe et qu'il n'est pas accessible en lecture :

```
$ [ -e confidentiel -a ! -r confidentiel ]; echo $?
1
```

Le code retour 1 indique que le test est faux. Une formule logique A *ET* B est fausse lorsque soit A est faux, soit B est faux. Dans notre cas, le résultat faux du test peut donc vouloir dire que :

- le fichier `confidentiel` n'existe pas
- le fichier `confidentiel` existe mais il est accessible en lecture

Pour en avoir le cœur net, on peut formuler le test « le fichier confidentiel existe ET n'est pas accessible en lecture OU BIEN il n'existe pas », ce qui se traduit par la commande :

```
$ [ -e confidentiel -a ! -r confidentiel -o ! -e confidentiel ]; echo $?
0
```

Exercice 4.2.2: Écrire un script `nomA.sh` qui retourne comme code retour 0 si son premier argument est une chaîne qui commence par la lettre majuscule A et 1 sinon. S'il n'y a pas d'argument, le script ne doit pas provoquer d'erreur et avoir un code retour 1. Attention à la priorité des opérateurs logiques.
Exemples d'exécution :

```
sh-3.2$ ./nomA.sh ; echo $?
1
sh-3.2$ ./nomA.sh Anatole ; echo $?
0
sh-3.2$ ./nomA.sh Bernard ; echo $?
1
sh-3.2$ ./nomA.sh albert ; echo $?
1
sh-3.2$ ./nomA.sh A ; echo $?
0
sh-3.2$ ./nomA.sh B ; echo $?
1
sh-3.2$ ./nomA.sh Andre Bernard ; echo $?
0
```

Solution page 224

À propos de l'utilisation de variables dans la commande `test`

Selon les opérateurs utilisés, la commande `test` attend deux ou trois arguments (l'opérateur en étant lui-même un). Lorsqu'on effectue un test sur une variable, si celle-ci n'existe pas (par exemple si on fait une faute de frappe) la substitution de valeur avec `$` va remplacer la variable par un vide et il manquera un argument à la commande `test`.

L'exemple suivant illustre cela :

```
$ mot1=alice
$ echo $mot1
$ [ $mot \> bob ]; echo $?
bash: [: >: unary operator expected
```

```
2
```

La variable définie est `mot1`. Comme on a frappé par inadvertance `mot` dans la commande `test`, on obtient un message d'erreur qui indique que Bash attend un opérateur unaire à la place de l'opérateur `>`. En effet, après substitution de `$mot`, le test réalisé est équivalent à `[ \> bob ]`. Comme l'expression est incomplète, elle produit une erreur d'exécution.

On peut éviter ce genre d'erreur, il est conseillé de mettre les variables en inhibition partielle (en l'entourant avec des guillemets). Avec l'ajout de l'inhibition partielle à la variable, si la variable n'existe pas, une chaînes de caractères vide est retournée par la substitution de la variable à la place d'un vide.

```
$ [ "$mot" \> bob ]; echo $?
1
```

Après substitution, ce dernier test est équivalent à `[ "" \> bob ]` et le code retour 1 indique que la chaîne `""` n'est pas supérieure à la chaîne `bob`

✍ **Challenge C42Q1**

Dans l'ordre alphanumérique

✍ **Challenge C42Q2**

Fichier protégé

3 La commande de test étendu

Nous avons vu que la commande `test` peut aussi s'écrire en encadrant le test à réaliser entre crochets simples `[ ... ]`. Il existe une version étendue de cette commande qui utilise des crochets doubles de cette manière `[[ ... ]]`. L'idée de cette commande est d'accepter des expressions écrites comme les autres langages de programmation. Pour cela, certains caractères du Bash perdent leur signification spéciale. Il s'ensuit que cette commande apporte quelques facilités d'écriture par rapport à la commande `test` :

— on peut utiliser les opérateurs `<` et `>` sans les faire précéder du caractère d'inhibition de caractère ('\').

```
$ [[ "bateau" < "navire" ]]; echo $?
0
```

— les opérateurs logiques `-a` et `-o` sont remplacés par les opérateurs et ('`&&`') et ou ('`||`') du langage de programmation C.

```
$ [[ -e confidentiel && ! -r confidentiel ]]; echo $?
1
```

— les parenthèses `(...)` peuvent être utilisées sans les faire précéder du caractère d'inhibition de caractère (les parenthèses ne servent plus à lancer un sous-shell pour exécuter ce qui est entre les parenthèses).

```
$ [[ -x justexecutable &&  ! (  -r justexecutable || -w justexecutable ) ]];
    echo $?
0
```

— les expressions arithmétiques sont évaluées à l'intérieur des `[[ ... ]]`.

```
$ a=2
$ b=4
$ [[ $a*3 -gt 10 ]]; echo $?
1
$ [[ $a*3 -eq $b+2 ]]; echo $?
0
```

— lorsqu'on utilise l'opérateur `==` ou l'opérateur `!=`, la chaîne de caractères à droite de l'opérateur est comprise comme un motif. Dans ce cas, ce n'est pas l'égalité ou la différence que l'on teste mais le filtrage par motif (*pattern matching*). Le filtrage par motif suit la même syntaxe que celle utilisée par la substitution de nom de fichier (voir l'activité 2.2). Pour rappel, dans un motif, le caractère spécial `*` remplace n'importe quelle séquence de caractères (éventuellement aucun), le caractère spécial `?` remplace n'importe quel caractère exactement une fois et les crochets permettent de remplacer exactement un des caractères indiqués entre les crochets.

```
$ [[ titu == t?t? ]]; echo $?
0
$ [[ taratata == t*a ]]; echo $?
0
$ [[ titu == t[aeiou]t[aeiou] ]]; echo $?
0
```

— Enfin, l'opérateur `=~` vérifie si la chaîne de caractères à gauche de l'opérateur vérifie l'expression régulière à droite. La syntaxe d'une expression régulière est la même que celle utilisée par la commande `grep` étendue. Ici, c'est un filtrage par expression régulière qui est testé. Attention, les caractères spéciaux utilisés dans le filtrage par motif sont différents de ceux utilisés par expression régulière. Le tableau 1 de l'activité 3.3 liste les caractères spéciaux des expressions régulières.
À noter que la chaîne de caractères doit être écrite entre des doubles quotes et non l'expression régulière. Le format d'écriture est alors le suivant :

```
[[ "string" =~ regex ]]
```

Par exemple, pour vérifier que la variable nombre contient bien un nombre entier, nous écrirons :

```
$ nombre=210
$ [[ "$nombre" =~ ^[+-]?[[:digit:]]+$ ]]; echo $?
0
```

L'expression régulière indique qu'une chaîne de caractères est un nombre si elle commence éventuellement par un signe, et qu'elle comporte au moins un chiffre et uniquement des chiffres.

Exercice 4.2.3: Récrire le script `nomA.sh` de l'exercice précédent en utilisant la commande de test étendue.

Solution page 224

✍ **Challenge C42Q3**

X et Y

4 Effectuer des tests numériques avec la commande `let`

Nous avons vu dans l'activité 3.4 que la commande `let` qui s'abrège par l'écriture `(( ... ))` permet de réaliser des calculs avec des expressions arithmétiques et en particulier des tests arithmétiques. Cette commande n'affiche rien mais son code retour peut être utilisé pour réaliser des tests.

Dans le cas de tests arithmétiques, si l'expression entre les doubles parenthèses est vraie le code retour vaut 0 et il vaut 1 sinon. Ce qui est équivalent aux résultats obtenus avec la commande `test`.

```
$ a=2
$ b=4
$ [ $a -lt $b ]; echo $?
0
$ (( a < b )); echo $?
0
$ [ $a -eq $b ]; echo $?
1
$ (( a == b )); echo $?
1
```

Attention cependant si l'expression arithmétique n'est pas une expression conditionnelle, l'interprétation du code retour est délicate. En effet, le code retour obtenu est 0 (vrai) si la valeur de l'expression entre les doubles parenthèses donne une valeur différente de 0 et 1 (faux) si la valeur de l'expression vaut 0.
Ainsi le résultat d'une expression arithmétique non conditionnelle dont on utilise le code retour est équivalent à faux si la valeur de l'expression est 0 et vrai autrement. **Il est donc important de ne pas confondre le code retour et la valeur de l'expression entre parenthèses**.

```
$ a=4
$ (( a-4 )); echo $?
1
$ (( a-2 )); echo $?
0
$ (( a/10 )); echo $?
1
$ (( a/2 )); echo $?
0
```

La valeur de l'expression arithmétique $a - 4$ est 0 donc le code retour de `(( a-4 ))` est 1, ce qui, utilisé comme un test, est équivalent à faux. L'interprétation serait donc « a est-il différent de 4 ? ». De la même manière l'interprétation du code retour de `(( a-2 ))` serait « a est-il différent de 2 ? » ; et pour `(( a/10 ))` : « la division entière de a par 10 est-elle différente de 0 ? » ou autrement dit « a est-il supérieur ou égal à 10 ? ».

5 Conclusion

Dans cette activité, nous avons vu comment utiliser la commande `test` pour comparer des valeurs arithmétiques, comparer des chaînes de caractères et comment effectuer des tests sur des fichiers. Au besoin il est possible de combiner des tests avec des opérateurs logiques. L'interprétation de la commande `test` se fait à travers son code retour qui vaut 0 si le test est vrai et 1 si le test est faux.

Le Bash fournit une version étendue de la commande `test` qui offre entre autre la possibilité de comparer des chaînes de caractères avec des motifs.

Enfin, il est aussi possible d'utiliser le code retour de la commande `(( ... ))` pour effectuer des tests numériques mais selon l'expression testée l'interprétation peut être délicate.

Ces commandes élémentaires nous permettent d'effectuer des tests mais, exécutées seules, elles ont un intérêt limité. Nous allons voir dans les activités suivantes que combinées avec les structures de contrôle fournies par le Bash, elles permettent de contrôler l'enchaînement de commandes.

Solutions des exercices

Solution de l'exercice 4.2.1 page 216:
Le nombre d'arguments d'un script est contenu dans la variable `$#`. Il s'agit donc de tester que la valeur numérique de cette variable est supérieure à 1 (ou supérieure ou égale à 2). Le code retour du script est le code retour de la dernière commande exécutée, ici ce sera le code retour du test donc 0 si le test est vrai et 1 sinon.
Attention de bien comparer le contenu de cette variable de manière numérique et pas comme une chaîne de caractères. Et faites bien attention aux espaces obligatoires dans le test.

```
#!/bin/bash
# aumoinsdeux.sh : teste que le nombre d'arguments est au moins 2
[ $# -gt 1 ]
```

ou bien

```
#!/bin/bash
# aumoinsdeux.sh : teste que le nombre d'arguments est au moins 2
[ $# -ge 2 ]
```

Solution de l'exercice 4.2.2 page 219:
Les chaînes pour lesquelles le test doit être vrai doivent commencer par la lettre majuscule `A`, c'est-à-dire celles qui viennent après la chaîne `A` dans le codage ASCII. La plus petite chaîne qui vient après `A` et qui ne commence pas par un `A` est la chaîne `B`.
Le script doit donc tester que le premier argument est supérieur ou égal à `A` et inférieur à `B`.
Nous n'avons pas à disposition d'opérateur $\leq$, il faut donc tester si l'argument est égal à la lettre `A` ou bien s'il est supérieur à `A` et inférieur à `B`. Comme le ET est prioritaire sur le OU, il faut commencer par le test d'égalité.
Enfin, pour éviter que le script ne produise une erreur au cas où il n'y a aucun argument, il faut prendre soin d'encadrer `$1` de guillemets afin que la substitution donne une chaîne vide.

```
#!/bin/bash
# nomA.sh : teste si le premier argument est une chaine qui commence par A
[ "$1" = A -o "$1" \> A -a  "$1" \< B ]
```

Solution de l'exercice 4.2.3 page 221:
On utilise l'opérateur `==` pour effectuer un filtrage par motif : un nom qui commence par la lettre majuscule `A` est une chaîne donc le premier caractère est `A` suivi d'une séquence de caractères quelconques, éventuellement vide.

```
#!/bin/bash
# nomA.sh : teste si le premier argument est une chaine qui commence A
[[ "$1" == A* ]]
```

Activité 4.3

Structures conditionnelles

1 Introduction

Prendre des décisions, conditionner l'exécution d'une commande au résultat d'une autre sont des éléments indispensables dès lors qu'on souhaite programmer des tâches un peu complexes.

Dans l'activité précédente nous avons vu comment effectuer des tests sur des chaînes de caractères, sur des nombres et sur des fichiers. Ces tests sont effectués à l'aide de commandes et c'est leur code retour qui indique si le test est vrai ou faux.

Dans cette activité nous allons voir comment utiliser le code retour d'une commande pour conditionner l'exécution d'autres commandes. Il s'agit de voir les structures de contrôle conditionnel du Bash : l'enchaînement conditionnel de commandes, la commande `if` et la commande `case`.

2 Enchaînement conditionnel

L'enchaînement conditionnel de commandes consiste à conditionner l'exécution d'une commande au résultat d'une autre. Selon le code retour d'une première expression, une autre sera ou ne sera pas exécutée.

Il existe trois caractères spéciaux permettant l'enchaînement de commande : le point-virgule, la double barre verticale et la double esperluette (aussi appelé « et commercial »).

2.1 Le point-virgule

commande1 ; *commande2*

Comme nous l'avons vu dans l'activité 2.5, le point-virgule est un séparateur de commande et il permet d'exécuter plusieurs commandes sur une même ligne de commande. Ces commandes sont exécutées les unes après les autres sans se soucier de la réussite ou de l'échec de chacune d'elles. Cet enchaînement n'est à utiliser que lorsqu'on est sûr de la réussite de chacune des commandes ou bien lorsque leur éventuel échec n'a aucune incidence sur la suite.

On peut le voir comme un moyen de regrouper plusieurs lignes de commandes sur une même ligne de commande. Par exemple, si on veut créer un répertoire et l'utiliser comme répertoire courant pour y créer un fichier vide, on pourra écrire :

```
$ mkdir ~/repertoire1; cd ~/repertoire1; touch fichiervide
```

Cette ligne de commande peut également s'écrire en 3 lignes de commandes.

2.2 La double esperluette

commande1 `&&` *commande2*

Avec cet enchaînement la *commande2* n'est exécutée que si le code retour de la *commande1* est 0. Autrement dit si la *commande1* réussit.

Par exemple on peut tester l'existence d'un fichier avant d'en afficher le contenu :

```
$ [ -f monCV ] && cat monCV
```

Si le fichier `monCV` existe la commande `[ -f monCV ]` a comme code retour 0, soit « vrai », et `cat monCV` sera exécutée mais si le fichier n'existe pas elle sera ignorée.

2.3 La double barre verticale

commande1 `||` *commande2*

Cette fois la *commande2* ne sera exécutée que si la *commande1* échoue ; par exemple si c'est un test dont le résultat est « faux ».

On peut compléter l'exemple précédent pour qu'un message s'affiche sur la sortie d'erreur au cas où le fichier `monCV` n'existe pas :

```
$ [ -f monCV ] && cat monCV || echo "le fichier monCV n'existe pas" >&2
```

Écriture sur la sortie d'erreur

Pour rediriger un affichage vers le canal de sortie d'erreur dont le numéro est 2, on utilise la syntaxe `>&2`. Il ne faut pas mettre d'espace entre `>` et `&2`.

Attention les commandes enchaînées sont exécutées dans le sens d'écriture, de gauche à droite sans qu'il y ait de priorité. Le code retour d'un enchaînement est celui de la dernière commande exécutée.

Ainsi dans l'exemple précédent :

- si `monCV` existe, le résultat de `[ -f monCV ]` est 0 ce qui provoque l'exécution de la commande `cat monCV`. Cette dernière réussit, son code retour est 0 et par conséquent la commande `echo` est ignorée
- si `monCV` n'existe pas, le résultat de `[ -f monCV ]` est 1, la commande `cat monCV` ne s'exécute pas, le résultat de l'enchaînement `[ -f monCV ] && cat monCV` est le code retour de la seule commande qui a été exécutée : `[ -f monCV ]`, c'est-à-dire 1. Par conséquent la commande `echo` qui suit `||` sera exécutée et le message s'affichera.

Exercice 4.3.1: Étant donnée une variable `fich` qui contient un nom de fichier, quel enchaînement de commandes permet de lister le contenu de `fich` si c'est un répertoire ou bien d'afficher le message "Ca n'est pas un repertoire" ?
Faire en sorte de ne pas produire d'erreur si la variable `fich` n'existe pas.

Solution page 233

Exercice 4.3.2: Écrire un script shell de nom `echo2.sh` qui a le comportement suivant :
- s'il n'y a pas d'argument, ou s'il n'y en a qu'un, le script affiche sur la sortie d'erreur le message "Il me faut deux arguments" et s'arrête avec un code retour 1 ;
- s'il y a plus de deux arguments, le script affiche sur la sortie d'erreur le message "Il y a trop d'arguments" et s'arrête avec un code retour 2 ;
- s'il y a deux arguments, le script les affiche sur la sortie standard et termine avec un code retour 0.

Solution page 233

✍ **Challenge C43Q1**

Oui ou non et code retour

3 La structure de contrôle conditionnel `if`

L'enchaînement de commandes est pratique mais se limite à l'exécution conditionnelle de commandes simples et peut vite devenir illisible si on enchaîne plusieurs commandes.

La commande `if` permet une écriture à la fois plus lisible et des contrôles plus élaborés. Il s'agit de conditionner l'exécution d'une suite de commandes au résultat d'une condition.

3.1 Condition simple

Dans sa version la plus simple la structure conditionnelle permet d'effectuer une opération si une condition est réalisée, la syntaxe est la suivante :

```
if condition
then
    lignes-commandes-si-vrai
fi
```

Attention : pour éviter toute erreur dans l'écriture de la structure il est nécessaire de respecter les passages à la ligne. On écrit le mot clé `if` suivi de la condition sur une seule ligne puis on passe à la ligne, on écrit le mot clé `then` seul sur une ligne, on passe de nouveau à la ligne pour écrire les lignes de commandes à exécuter lorsque la condition est vraie, puis on termine la structure en écrivant le mot clé `fi` seul sur une ligne.

L'interprétation de cette structure de contrôle est la suivante : si, et seulement si, la *condition* est vérifiée, alors toutes les commandes entre le `if` et le `fi` seront exécutées.

Pour que la condition soit vérifiée il faut que le code retour de celle-ci soit 0 (interprété comme vrai). Le plus souvent la condition est exprimée à l'aide de la commande `test` vue dans l'activité précédente mais cela peut aussi être n'importe quelle commande sachant que c'est son code retour qui sera utilisé pour décider de l'exécution des *lignes-commandes-si-vrai*.

Exemple

À titre d'exemple on peut tester la valeur d'une variable avant d'effectuer un calcul :

```
if test "$mavariable" -ne 0
then
```

```
    resultat=$((10 / mavariable))
fi
```

Qu'on peut aussi écrire en utilisant la commande `(( ))` à la place de la commande `test`

```
if (( $mavariable != 0 ))
then
    resultat=$((10 / mavariable))
fi
```

3.2 Condition avec alternative

La structure conditionnelle peut inclure une partie `else` qui permet d'exécuter une suite de commandes alternatives au cas où la condition n'est pas vérifiée. Dans ce cas la syntaxe est :

```
if condition
then
    lignes-commandes-si-vrai
else
    lignes-commandes-si-faux
fi
```

L'interprétation est cette fois : si, et seulement si, la *condition* est vérifiée, alors les *lignes-commandes-si-vrai* seront exécutées sinon les *lignes-commandes-si-faux* seront exécutées.

Exemple

On peut ainsi récrire l'exemple de la section précédente avec le fichier `monCV` en utilisant une structure conditionnelle :

```
if [ -f monCV ]
then
    cat monCV
else
    echo "le fichier monCV n'existe pas" >&2
fi
```

3.3 Conditions imbriquées

Parfois une simple alternative ne suffit pas à traduire ce qu'on veut programmer, en particulier si plusieurs conditions entrent en jeu. Heureusement il est possible d'imbriquer des structures conditionnelles à l'intérieur d'autres structures conditionnelles pour obtenir des constructions telles que :

```
if condition-1
then
    lignes-commandes-si-vrai-1
else
    if condition-2
    then
        lignes-commandes-si-vrai-2
```

```
    else
        if condition-3
        then
            ...
        fi
    fi
fi
```

Pour simplifier l'écriture de telles structures imbriquées le Bash permet de contracter un `else` suivi d'un `if` avec le mot-clé `elif` et de terminer les imbrications par un seul `fi`.
On peut donc écrire plus simplement :

```
if condition-1
then
    lignes-commandes-si-vrai-1
elif  condition-2
then
    lignes-commandes-si-vrai-2
elif condition-3
then
    ...
fi
```

Exemple

Pour illustrer les imbrications de structures conditionnelles, écrivons un script dont le traitement consiste à écrire la phrase « I love Bash » à la fin d'un fichier qui s'appelle `message`.

Pour écrire ce script, qu'on nommera `lovebash.sh`, nous devons faire plusieurs vérifications : est-ce que le fichier `message` existe, est-ce un fichier ordinaire et est-il accessible en écriture. Si toutes les conditions sont réunies on ajoute la phrase « I love Bash » à la fin de ce fichier en utilisant la redirection `>>` et s'il n'existe pas on le crée en redirigeant le résultat de la commande `echo` dans le fichier avec `>`.
Écrivons déjà cela :

```
#!/bin/bash
if [ -f message -a -w message ]
then
    echo "I love Bash" >> message
else
    echo "I love Bash" > message
fi
```

Tel qu'il est écrit, ce script va tenter de créer ou d'écraser le fichier message si le test est faux. Mais ce test peut être faux pour plusieurs raisons : soit le fichier n'est pas un fichier ordinaire, soit le fichier est un fichier ordinaire mais il n'est pas accessible en écriture, soit il n'existe tout simplement pas. Or si ça n'est pas un fichier ordinaire ou bien s'il n'est pas accessible en écriture la commande dans la partie `else` va provoquer une erreur.
Nous pouvons compléter ce script en imbriquant des `elif` pour détailler tous les cas : si le message n'existe pas on le crée ; si c'est un fichier ordinaire mais sans droit d'écriture, on modifie les droits et on écrit dedans ; sinon on affiche un message sur la sortie standard d'erreur.

```
#!/bin/bash
```

```
if [ -f message -a -w message ]
then
    echo "I love Bash" >> message
elif [ ! -e message ]
then
    echo "I love Bash" > message
elif [ -f message -a ! -w message ]
then
    chmod u+w message
    echo "I love Bash !" >> message
else
    echo "message existe mais n'est pas un fichier ordinaire" >&2
fi
```

Exercice 4.3.3: Écrire un script shell de nom `testsomme.sh` qui attend deux ou trois entiers en arguments et qui a le comportement suivant :

- s'il n'y a pas deux ou trois arguments, le script écrit sur la sortie d'erreur le message : `Donnez 2 ou 3 arguments` et s'arrête avec un code retour 1 ;
- s'il y a deux arguments, le script affiche sur la sortie standard la somme des deux arguments accompagnée du message : `La somme vaut`
- s'il y a trois arguments, le script calcule la somme des deux premiers, la compare au troisième et, selon le résultat, affiche l'un des messages suivants : `Somme egale`, `Somme inferieure` ou `Somme superieure`.

Rq : Si on ne donne pas comme argument des entiers, il y aura une erreur à l'exécution du script mais nous n'en tiendrons pas compte ici.

Solution page 233

✍ **Challenge C43Q2**

Tester les arguments d'un script

4 La structure de branchement conditionnel case

L'instruction `case` est une bonne alternative à l'utilisation de `if` imbriqués, en particulier quand il s'agit de tester une valeur et d'exécuter différentes commandes en fonction de cette valeur. La syntaxe de l'instruction `case` est la suivante :

```
case expression in
motif1)
    lignes-commandes1;;
motif2)
    lignes-commandes2;;
...
motifN)
    lignes-commandesN;;
esac
```

Notez les particularités de cette syntaxe :

- l'expression à tester est encadrée des mots-clés `case` et `in` sur une seule ligne

— suivent ensuite chacun des cas introduits par un motif suivi d'une parenthèse fermante `)` puis de la liste des commandes à effectuer terminée par un double point-virgule `;;`
— enfin la structure est fermée par le mot clé `esac` seul sur une ligne.

On peut traduire cette structure de contrôle par :

— au cas où la valeur de *expression* est de la forme *motif1*, exécuter les *lignes-commandes1* ;
— au cas où la valeur de *expression* est de la forme *motif2*, exécuter les *lignes-commandes2* ;
— ...
— au cas où la valeur de *expression* est de la forme *motifN*, exécuter les *lignes-commandesN*.

L'exécution d'un `case` se déroule de la manière suivante : d'abord l'*expression* est évaluée afin d'obtenir une chaîne de caractères. Celle-ci est comparée avec chaque motif, dans l'ordre. Au premier motif qui correspond les *lignes-de-commandes* sont exécutées jusqu'au `';;'` puis le `case` s'arrête en retournant comme code retour celui de la dernière commande exécutée.

Ainsi la structure `case` exécute les commandes en fonction du filtrage de la valeur de l'*expression* selon des motifs. Les motifs sont décrits avec la même syntaxe que celle des abréviations pour le nom des fichiers (voir l'activité 2.2). Les caractères spéciaux pouvant être utilisés sont définis dans le tableau 2 de l'activité 2.2.

La plupart du temps l'expression est une substitution de variable (on veut tester le contenu de la variable) ou le résultat de la substitution d'une commande ou d'une expression numérique (dans ce dernier cas c'est la chaîne de chiffres qui est testée).

Si aucun des motifs ne correspond à la valeur de l'expression alors aucune commande n'est exécutée et le code retour de l'instruction `case` est 0. Toutefois s'il y a besoin d'exécuter des commandes dans le cas où aucun des motifs ne correspond on peut utiliser le motif `*` comme cas par défaut puisque celui-ci filtre n'importe quelle chaîne de caractères.

Normalement si la structure est bien écrite les différents cas doivent être disjoints, c'est-à-dire que pour une valeur donnée de l'expression un seul motif doit correspondre. Si ça n'est pas le cas seules les commandes du premier motif correspondant à la valeur de l'expression seront exécutées.

Si plusieurs cas doivent mener à l'exécution d'une même liste de commandes on peut écrire plusieurs motifs séparés par une barre verticale :

```
case expression in
motif1 | motif2 | motif3)
    lignes-commandes123;;
...
esac
```

Exemple

Le script suivant écrit le type d'un fichier donné en argument en se fondant sur l'extension du nom de fichier. On utilise le caractère spécial `*` dans les filtres pour ignorer le nom du fichier jusqu'à l'extension. Le troisième cas est constitué de trois filtres séparés par une barre verticale. Le script commence par vérifier qu'il y a bien exactement un argument. Si ce n'est pas le cas, un message d'erreur et l'usage du script sont affichés dans la sortie d'erreur (numéro de canal 2).

```
#!/bin/bash
if [ $# -ne 1 ]
then
    echo "Erreur : nombre d'argument incorrect" >&2
    echo "Usage : $0 fich"  >&2
    exit 1
fi
```

```
case $1 in
    *.c)
        echo "Code source en langage c";;
    *.sh)
        echo "Script bash";;
    *.jpeg | *.jpg | *.png)
        echo "Image";;
    *)
        echo "Type de fichier non reconnu";;
esac
```

Ce deuxième exemple demande à l'utilisateur de répondre par oui ou non à une question donnée en argument (à condition qu'il y en ait une). Il prévoit toutes les formes de réponses possibles pour oui ou non. La réponse est lue sur l'entrée standard par la commande `read` et placée dans la variable qui suit la commande.

```
#!/bin/bash
if [ $# -eq 1 ]
then
    echo "$1 ?"
    echo "Etes vous d'accord (oui ou non) ?"
    read rep
    case $rep in
    [oO] | [oO][uU][iI])
        echo "Ok, merci"
        ;;
    [nN] | [nN][oO][nN])
        echo "Tres bien, je respecte votre choix"
        ;;
    *)
        echo "Desole, je ne comprends pas votre reponse"
        ;;
    esac
fi
```

✍ **Challenge C43Q3**

Une touche

5 Conclusion

Dans cette activité nous avons vu comment contrôler l'exécution des commandes dans un script de manière à ce que différents traitements puissent être réalisés en fonction du code retour d'une commande. Nous avons d'abord vu l'enchaînement conditionnel de commandes qui assujettit l'exécution d'une commande à la réussite ou à l'échec de la commande précédente. La structure de contrôle conditionnel `if` permet de décider quelles actions réaliser en fonction du résultat d'un test (sur des nombres, des chaînes de caractères ou des fichiers) ou du code retour d'une commande. La commande `case` quant à elle est une structure de branchement conditionnel qui permet de comparer une chaîne de caractères avec plusieurs motifs afin de décider de la suite des commandes à exécuter. Toutes ces structures sont essentielles pour prendre des décisions et contrôler le déroulement de l'exécution d'un script.

Solution des exercices

Solution de l'exercice 4.3.1 page 226:

```
[ -d "$fich" ] && ls "$fich" || echo "$fich n'est pas un repertoire"
```

On encadre la substitution de la variable `fich` par des guillemets afin de ne pas produire d'erreur si celle-ci est vide. Si elle contient bien un nom de répertoire le premier test est vrai et l'enchaînement `&&` se poursuit avec la commande `ls`. Si le test échoue la deuxième commande n'est pas exécutée et c'est la commande située après l'enchaînement `||` qui est exécutée.

Solution de l'exercice 4.3.2 page 227:

```
#!/bin/bash
# echo2.sh : affiche ses arguments s'il y en a exactement 2
[ $# -lt 2 ] && echo "Il me faut deux arguments">&2 && exit 1
[ $# -gt 2 ] && echo "Il y a trop d'arguments">&2 && exit 2
echo $1 $2
```

Premier enchaînement : si le test est vrai la deuxième commande est exécutée et comme `echo` a toujours un code retour 0 la troisième commande est aussi exécutée. Cette dernière termine le script avec un code retour 1.

Le deuxième enchaînement ne sera exécuté que si le premier n'a pas été jusqu'au bout (donc le script ne s'arrête pas), ce qui arrive uniquement si le test `[ $# -lt 2 ]` est faux. Il y a donc au moins deux arguments. Cet enchaînement se poursuit après le test `[ $# -gt 2 ]` uniquement s'il est vrai ; dans ce cas les deux commandes suivantes s'enchaînent et le script termine avec un code retour 2 après avoir écrit sur la sortie d'erreur le message "Il y a trop d'arguments".

La troisième ligne de commande du script ne sera exécutée que si aucun des deux enchaînements précédents n'est allé jusqu'au bout. Cela signifie que les deux tests sont faux et qu'il y a donc exactement 2 arguments. Il suffit de les afficher et le script termine avec le code retour de la commande `echo` qui est 0.

Solution de l'exercice 4.3.3 page 230:

```
#!/bin/bash
# testsomme.sh
# Attend 2 ou 3 arguments sinon affiche le message d'erreur "
# Si deux arguments, affiche leur somme
# Si 3 arguments : si $1+$2 = $3 affiche "Somme egale"
#                  sinon si $1 + $2 > $3 affiche "Somme superieure"
#                  sinon affiche "Somme inferieure"

# S'il y a moins de 2 ou plus de 3 arguments
# on arrete le script avec un message d'erreur
if [ $# -lt 2 -o $# -gt 3 ]
then
    echo "Donnez 2 ou 3 arguments">&2
    exit 1
fi
```

```
# A partir d'ici on sait qu'il y a 2 ou 3 arguments
if [ $# -eq 2 ]
then
    echo "La somme vaut $(($1 + $2))"
else
    # a partir d'ici il y a forcement 3 arguments
    # on calcule la somme des deux premiers
    som=$(($1 + $2))
    # et on la compare au troisieme
    if [ "$som" -eq $3 ]
    then
        echo "Somme egale"
    elif [ "$som" -gt $3 ]
    then
        echo "Somme superieure"
    else
        echo "Somme inferieure"
    fi
fi
```

Activité 4.4

Structures itératives

1 Introduction

Dans cette partie nous allons nous concentrer sur les structures itératives. Ce sont des commandes qu'on appelle plus simplement des "boucles". Dans une boucle, un même bloc de commandes s'exécute plusieurs fois. On écrit une seule fois le bloc de commandes à répéter au sein de la boucle, et ce bloc sera ensuite exécuté un certain nombre de fois. On dit que la boucle fait plusieurs tours. Le nombre de tours qui sera réalisé est déterminé par le type de boucle utilisé et les paramètres spécifiés pour celle-ci. Il existe plusieurs types de boucles. Nous allons étudier ici les deux plus courantes. Il s'agit :

- de l'itération conditionnelle ; elle continue de boucler sur le bloc de commandes tant que la condition de continuation est vérifiée.
- de l'itération bornée ; elle réalise l'exécution du bloc de commandes pour chaque élément d'une liste.

2 Rappel sur les expressions conditionnelles

L'itération conditionnelle repose sur une expression conditionnelle. Cette expression est évaluée sous la forme d'un test. Aussi il est important de bien connaître les termes des expressions composées en Bash. Dans ce paragraphe, nous allons revoir les points importants présentés dans l'activité 4.2.

En shell, le test d'une expression retourne un code retour qui vaut 0 pour vrai et qui vaut une valeur différente de 0 pour faux. Pour ceux parmi vous qui ont l'habitude d'autres langages de programmation il faut être très vigilant à ce niveau, car généralement dans les autres langages la signification du vrai et du faux est inversée.
Pour le shell une commande qui renvoie 0 signale que son exécution s'est bien passée (qu'il n'y a pas eu d'erreur), donc que tout est OK. Il y a qu'un cas normal d'exécution, le cas sans erreur. Par contre, le cas d'exécution de la commande avec erreur peut être du à une multitude de cause. Pour refléter cet état de fait, le code retour d'une commande vaut 0 avec la signification "vrai" et autre chose que 0 pour celle de faux.

Parmi les commandes existantes, il y a la bien nommée commande : `test`. C'est une commande qui permet d'exprimer un test sous la forme d'une expression booléenne. Pour cela il suffit de passer en argument de cette commande l'expression booléenne à évaluer :

```
$ test expression_booléenne
```

Il existe une variante syntaxique de cette commande qui s'écrit avec des crochets : `[ ]`, entre lesquels on écrit l'expression booléenne :

```
$ [ expression_booléenne ]
```

Pour cette écriture attention de bien **laisser une espace** après le crochet ouvrant et avant le crochet fermant ! Pour ce qui est de la syntaxe des expressions booléennes, comme vous l'avez vu dans l'activité 4.2, elles emploient principalement des opérateurs à base de lettres. Les opérateurs courants sont rappelés par le tableau 1.

Opérateur	Sémantique	Syntaxe
Opérateurs de logiques		
`-a`	et logique (and)	*expr1* `-a` *expr2*
`-o`	ou logique (or)	*expr1* `-o` *expr2*
`!`	négation logique (not)	`!` *expr*
Pour les valeurs numériques		
`-eq`	égale à (equal)	*expr1* `-eq` *expr2*
`-ne`	différente de (not equal)	*expr1* `-ne` *expr2*
`-lt`	inférieure à (less than)	*expr1* `-lt` *expr2*
`-gt`	supérieure à (greater than)	*expr1* `-gt` *expr2*
`-le`	inférieure ou égale à	*expr1* `-le` *expr2*
`-ge`	supérieure ou égale à	*expr1* `-ge` *expr2*
Pour les chaînes de caractères		
`=`	égale à (equal)	*chaîne1* `=` *chaîne2*
`!=`	différente de (not equal)	*chaîne1* `!=` *chaîne2*
`-z`	la chaîne a une taille égale à zéro	`-z` *chaîne*
Pour les fichiers		
`-e`	le fichier existe	`-e` *fichier*
`-x`	le fichier est exécutable	`-x` *fichier*
`-f`	le fichier est ordinaire	`-f` *fichier*
`-d`	le fichier est un répertoire	`-d` *fichier*
`-s`	le fichier est non vide	`-s` *fichier*

Tableau 1 – Opérateurs.

3 L'itération conditionnelle

Un premier type de boucle est celui consistant à répéter l'exécution du bloc de commandes tant qu'une condition est vérifiée. Cet usage est connu sous la boucle `while` qui est commun à beaucoup de langages de programmation.

Avec le Bash, la commande `while` va servir à répéter l'exécution d'une séquences de commandes tant que la condition après le `while` retourne un code retour à vrai. La condition peut être une commande quelconque ou l'évaluation d'une expression. Dans ce dernier cas, la syntaxe de la condition fait donc naturellement intervenir la commande `test`.

```
while condition
do
      commande1
      commande2
    ...
done
```

La condition à considérer s'écrit après le mot clé `while`, sur la même ligne. Vient ensuite la séquence des commandes pour constituer le corps de la boucle sur lequel les boucles vont être effectuées. Le corps de la boucle est délimitée par les mots clés `do` et `done`. Ces délimiteurs doivent chacun être isolés sur une ligne. Pour une écriture plus compacte, il est courant d'utiliser l'opérateur enchaînement séquentiel de commande `';'` pour mettre `while` et `do` sur une même ligne.

```
while condition  ; do
    commande1
    commande2
    ...
done
```

3.1 Exemples d'utilisation

Écrivons un petit script pour illustrer l'usage de cette boucle. Son traitement va consister à répéter le mot qui lui est fourni un certain nombre de fois en fonction d'un nombre indiqué. Nous allons donc appeler ce script `repeat.sh`. Il aura besoin du mot à répéter et du nombre de répétitions à faire. Ce nombre sera par défaut `10`. Ces deux paramètres seront indiqués comme argument lors du lancement du script.

Écrivons déjà la structure de la boucle `while`, en laissant en suspens le test pour le moment, et écrivons juste la commande à répéter entre les délimiteurs `do` et `done` :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
while ??? ; do
  echo  $mot
done
```

Maintenant que la structure est en place, il nous faut faire en sorte d'effectuer `10` tours dans cette boucle. Nous allons donc définir une variable `i` qui va nous servir à compter le nombre de tours. Au départ cette variable aura la valeur 0, et à la fin de chaque tour nous allons incrémenter sa valeur à l'aide de la syntaxe la substitution arithmétique `$(( ... ))` :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
i=0
while ??? ; do
  echo  $mot
  i=$(($i+1))
done
```

Grâce à cette information du nombre de tours, nous pouvons facilement déterminer le test qu'il nous faut écrire pour calibrer correctement notre boucle. Il va s'agir de comparer la valeur du compteur `i` avec le nombre de tours à réaliser que nous avons conservé dans la variable `nb`.

Tant que la valeur de `i` est inférieure à `nb` on continue. Cela s'écrit avec l'opérateur `-lt` (*less than*), qui veut dire "inférieure à" en français :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
```

```
i=0
while [ $i -lt $nb ] ; do
  echo  $mot
  i=$(($i + 1))
done
```

Voilà il ne nous reste plus qu'à enregistrer, fixer le droit d'exécution et tester le script :

```
$ chmod +x  repeat.sh
$ ./repeat.sh Bonjour 3
Bonjour
Bonjour
Bonjour
$
```

Exercice 4.4.1: Écrire un script de comptage appelé `count.sh`. Ce script affiche toutes les valeurs d'un intervalle indiqué par la borne inférieure et la borne supérieure. Les bornes sont indiqués par 2 arguments de la manière suivante :

```
$ count.sh 3 7
3
4
5
6
7
$
```

Si les arguments sont absents leurs valeurs par défaut doivent être respectivement 1 et 10.

Solution page 246

3.2 Lecture de fichier avec `while` et `read`

Une utilisation classique de la boucle `while` est la lecture de fichier ligne à ligne. Cette lecture se fait en utilisant la commande `read`. Mais prenons tout de suite un exemple avec le script suivant appelé `print.sh`.

```
#!/bin/bash
a=0
while read ligne; do
  ((a++))
  echo $a $ligne
done
```

Ce script lit une ligne depuis l'entrée standard, l'affiche précédée de son numéro de ligne et recommence tant que la fin de fichier (indiqué par ctrl+d) n'a pas été rencontrée. Il faut savoir que la commande `read` retourne le code vrai (0) quand la fin de fichier n'a pas été lue. Ainsi la condition d'arrêt de cette boucle est définie par la commande `read` quand elle lit la fin du fichier. A noter que la variable *ligne* prend à chaque itération de la boucle le contenu d'une nouvelle ligne lue de l'entrée standard. Dans la version présentée, ce script doit recevoir sur son entrée standard le contenu d'un fichier comme par exemple :

```
$ cat print.sh | ./print.sh
```

Si à la place d'un contenu, c'est un nom de fichier qui est donné en argument comme :

```
$ ./print.sh print.sh
```

La redirection d'entrée doit être utilisée, le code du script `print.sh` devient :

```
#!/bin/bash
a=0
while read ligne; do
  ((a++))
  echo $a $ligne
done <$1
```

La redirection prend place après le marqueur de fin de commande de la boucle `while` à savoir après le `done`. Le symbole `$1` représente le premier argument de la ligne de commande et donc le nom du fichier à lire en entrée de la boucle. Lançons le script `print.sh` avec comme argument le fichier `print.sh`. Nous obtenons :

```
$ ./print.sh print.sh
1 #!/bin/bash
2 a=0
3 while read ligne; do
4 ((a++))
5 echo $a $ligne
6 done <"$1"
```

Nous avons présenté ici une des utilisations de la commande `read` avec la boucle `while` pour lire une ligne de l'entrée standard. Utilisée avec plusieurs arguments la commande read permet aussi de découper la ligne lue : le premier argument capte le premier mot de la ligne, le deuxième argument le deuxième mot, etc. S'il y a moins d'arguments que de mots sur la ligne, le dernier argument capte tout le reste de la ligne. Nous ne rentrerons pas plus dans les détails de cette commande.

3.3 Contrôle de la boucle `while`

Nous avons vu dans les sections précédentes que la boucle `while` est contrôlée par une condition. Il est aussi possible d'introduire des ruptures de séquence dans le corps de la boucle avec les commandes `break` et `continue`. Les commandes `break` et `continue` sont le plus souvent employées avec la boucle `while`, mais elles peuvent aussi être utilisées avec une boucle `for`.

3.4 La commande `continue`

La commande `continue` renvoie en début de boucle. Les commandes du corps de la boucle après la commande `continue` ne sont pas exécutées. Prenons tout de suite un exemple que nous allons commenter ensuite. Soit le script suivant que l'on appellera `continue.sh`

```
#!/bin/bash
a=0
while [ "$a" -le 10 ]; do
 a=$(($a+1))
 if [ "$a" -eq 2 ] || [ "$a" -eq  8 ]
 then
   continue      # renvoie au debut de la boucle while
 fi
```

```
 echo -n "$a "   # Cette partie ne sera pas exécutée pour a=2 et a=8
done
```

Si on lance le script `continue.sh` on obtiendra :

```
$ ./continue.sh
1 3 4 5 6 7 9 10 11
```

Nous voyons dans le résultat que ni 2 ni 8 ne sont affichés. On peut voir la commande `continue` comme un retour direct à la ligne `while [ "$a" -le 10 ]; do`. Attention ici de ne pas créer une boucle infinie, ce qui peut arriver si la ligne `a=$(($a+1))` est placée après la commande `continue` puisque la variable a ne sera alors plus incrémentée. La commande `continue` peut prendre un paramètre numérique qui est utilisé en cas de boucles imbriquées. Ce paramètre indique alors le nombre de boucles imbriquées qu'il faut "remonter".

3.5 La commande break

La commande `break` termine la boucle en sortant directement de celle-ci. Prenons le script précédent et changeons la commande `continue` par la commande `break` pour constituer le script `break.sh`.

```
#!/bin/bash
a=0
while [ "$a" -le 10 ]; do
 a=$(($a+1))
 if [ "$a" -eq 2 ] || [ "$a" -eq  8 ]
 then
   break      # quitte directement la boucle while
 fi
 echo -n "$a "   # Cette partie ne sera pas exécutée pour a supérieur ou égal à 2
done
```

Si on lance le script `break.sh` on obtiendra :

```
$ ./break.sh
1
```

Nous voyons dans le résultat que la boucle s'arrête dès que la valeur de a est supérieure ou égale à 2. La commande `break` renvoie directement à la ligne `done`. La commande `break` peut aussi prendre en argument un niveau d'imbrication de boucle. Par exemple, `break 3` va "terminer" trois boucles imbriquées.

Avec la commande `break`, il est possible de faire une boucle avec plusieurs conditions de sortie ou d'avoir la condition de sortie dans le corps de la boucle. La structure générale d'un telle boucle est la suivante :

```
while true
do
 commande 1
 if condition de sortie
 then
   break      # quitte directement la boucle while
 fi
 commande 2
done
```

La commande `true` renvoie toujours un code retour vrai (0). La boucle while se transforme en une boucle infinie. La sortie de la boucle s'effectue par la commande `break`.

✍ **Challenge C44Q1**

Statistiques sur un sous-ensemble de données

4 L'itération bornée

Le deuxième type de boucle est celui consistant à répéter l'exécution de la séquence de commandes pour chacun des éléments d'une liste autrement dit la séquence de commandes est à répéter un certain nombre de fois. En shell, on utilise pour cela la commande `for`. On trouve l'équivalent de cette commande dans les autres langages de programmation, où elle implique généralement l'usage d'un compteur. En shell les choses sont un peu différentes, la commande `for` va en fait parcourir les éléments d'une liste.

La syntaxe de cette structure fait donc apparaître :

- La liste à parcourir : il s'agit simplement d'une suite de valeurs séparées par des espaces ou des sauts de lignes.
- Le nom de la variable : qui à chaque tour de boucles va recevoir successivement chacune des valeurs de la liste.
- Et la séquence de commandes à exécuter à chaque passage.

```
for variable in liste
do
  commande1
  commande2
  ...
done
```

La liste à parcourir peut être indiquée littéralement mais elle peut provenir suite à une substitution de variable, de commande ou de nom de fichier. Si la liste n'est pas spécifiée, c'est la liste des arguments de la ligne de commande qui est retenue par défaut.

4.1 Exemple d'utilisation

Pour mettre en pratique cette boucle `for`, nous allons faire un premier test simple avec un script de nom `testFor.sh` :

```
$ vi testFor.sh
```

Le code de ce script va consister à parcourir une liste de 3 noms avec une variable appelée `courant`. À chaque tour de boucle on va juste afficher la valeur courante de cette variable :

```
#!/bin/bash
for courant in Pierre Paul Jacques
do
  echo $courant
done
```

Notez qu'une liste d'éléments s'écrit simplement sous la forme d'une suite de valeurs (ici des prénoms) séparées par des espaces.

On enregistre le fichier, on fixe le droit d'exécution, et on lance le script :

```
$ chmod +x testFor.sh
$ ./testFor.sh
Pierre
Paul
Jacques
$
```

On constate que chacun des noms s'affiche bien l'un après l'autre.

4.2 Liste issue d'une variable

La liste à parcourir peut aussi être exprimée par la valeur d'une variable. Par exemple, avec `$@` qui contient la liste de tous les arguments de lancement du script. Nous allons justement faire un petit script avec une boucle `for` qui va parcourir les valeurs présentes dans cette variable `$@`. Cela va avoir pour effet d'afficher toutes les valeurs qui sont fournies en argument de lancement du script.

Nous allons appeler ce script `afficheArg.sh` :

```
$ vi afficheArg.sh
```

On va utiliser une variable `num` pour numéroter chaque argument. Dans la structure du `for`, nous allons utiliser une variable `var` pour parcourir les éléments de `$@`. À chaque tour de boucle, on affichera le numéro puis la valeur de `var` et on incrémentera `num` pour la prochaine itération.

```
#!/bin/bash
num=1
for  var  in  "$@" ; do
  echo  "argument $num = $var"
  ((num++))
done
```

On enregistre le fichier, on fixe le droit d'exécution, et on lance le script avec quelques arguments :

```
$ chmod +x afficheArg.sh
$ ./afficheArg.sh  yep  "hip hip"
argument 1 = yep
argument 2 = hip hip
```

✍ **Challenge C44Q2**

L'espace occupé par les photos d'Alice

4.3 Liste issue d'une commande

En choisissant bien la liste à parcourir on peut revenir aisément au fonctionnement plus classique d'une boucle *for* à base de compteur. En effet, il suffit simplement d'utiliser l'affichage produit par la commande `seq` pour générer la liste. Cette commande attend 2 nombres en argument, et elle va produire l'affichage des valeurs successives entre ces 2 nombres :

```
$ seq  1  5
1 2 3 4 5
$
```

On va mettre cette possibilité en pratique en reprenant notre script `repeat.sh`. On va modifier son code pour utiliser cette fois-ci un `for` au lieu d'un `while`. La variable `i` va maintenant être celle qui va parcourir la liste du `for`, et cette liste va être produit par la commande `seq` sur l'intervalle `1` à `nb`. Nous allons capturer cet affichage sous forme de valeur grâce à la substitution de commande, à l'aide de la syntaxe `$( ... )`.

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
for i in  $(seq 1 $nb) ; do
  echo  $mot
done
```

Avec ce type d'écriture du `for` on n'a plus besoin d'incrémenter le compteur `i` à chaque tour, car il va passer tout seul d'une valeur à l'autre à l'aide de la séquence générée par `seq`. On enregistre, et on peut tester que le script fonctionne aussi bien qu'avant :

```
$ ./repeat.sh Bonjour 3
Bonjour
Bonjour
Bonjour
$
```

Le Bash offre aussi un moyen de générer une liste au moyen du développement d'accolades. Cette construction syntaxique génère des chaînes de caractères. Par exemple `img.{pdf,png,jpg}` va produire `img.pdf img.png img.jpg`. Le développement d'accolades trouve aussi son intérêt pour produire des suites. La suite des nombres de 0 à 9 s'écrit `{0..9}`

```
$ echo {0..9}
0 1 2 3 4 5 6 7 8 9
```

Imaginons que vous voulez produire une suite avec des numéros non continus comme par exemple les numéros des différentes activités de ce cours.

```
$ echo A{1..4}{0..6}
A10 A11 A12 A13 A14 A15 A16 A20 A21 A22 A23 A24 A25 A26 A30 A31 A32 A33 A34 A35 A36
    A40 A41 A42 A43 A44 A45 A46
```

Si nous voulons produire l'affichage d'un tableau. Le script ci-dessous le construit rapidement pour vous avec le développement d'accolades et une boucle `for`.

```
#!/bin/bash
     echo "| Act. | Commentaire |"
for i in  A{1..4}{0..6} ; do
  echo  "| $i  |             |"
done
```

L'exécution produit :

```
| Act. | Commentaire |
| A10  |             |
| A11  |             |
...
| A45  |             |
| A46  |             |
```

4.4 Boucle avec un compteur

La boucle `for` peut aussi prendre une syntaxe héritée du langage de programmation C. Cette variante est caractérisée par trois paramètres qui contrôlent la boucle `for`. Ces trois paramètres sont `EXP1` : qui est l'initialisation de la boucle, `EXP2` qui présente le test qui conditionne l'exécution du prochain tour de boucle, et `EXP3` qui spécifie l'expression de comptage. EXP1, EXP2, et EXP3 sont des expressions arithmétiques.

```
for (( EXP1; EXP2; EXP3 ))
do
  command1
  command2
done
```

Cette structure de boucle est équivalente à

```
(( EXP1 ))
while (( EXP2 )); do
        command1
        command1
        (( EXP3 ))
done
```

Voyons un exemple ci dessous dans le script `for3.sh`.

```
#!/bin/bash
for ((i=1; i<=4; i++)); do
  echo "Nombre $i"
done
```

Dans le script `for3.sh`, le premier paramètre de la boucle `for` est la déclaration et l'initialisation d'une variable i à 1. Le deuxième paramètre est la condition d'entrée dans le prochain tour de boucle, dans notre cas i doit être inférieur ou égal à 4. Le troisième paramètre est l'expression de comptage qui à chaque passage de la boucle incrémente la valeur i de 1. Dans le corps de la boucle nous affichons simplement la valeur de i à chaque itération de la boucle. Le résultat de ce script est :

```
$ ./for3.sh
Nombre 1
Nombre 2
Nombre 3
Nombre 4
```

Exercice 4.4.2: Écrire le code d'un script de nom `pyramide.sh`, ce script prend en argument un entier qui correspond à la hauteur de la pyramide à afficher. La pyramide sera dessinée à l'aide de caractère O, en ajoutant un caractère à chaque étage, comme dans l'exemple ci-dessous :

```
$ pyramide.sh 5
O
OO
OOO
OOOO
OOOOO
$
```

Si l'argument est absent, sa valeur par défaut doit être 4.

Solution page 246

✍ **Challenge C44Q3**

HTML facile

5 Conclusion

Dans cette partie nous avons vu :

- à quoi sert une structure itérative
- que la commande `while` nous permet d'exécuter une même suite de commandes tant qu'une condition est vérifiée.
- que la commande `for` nous permet de parcourir les éléments d'une liste et d'exécuter une même suite de commandes pour chacun d'eux.
- et enfin qu'on peut créer une boucle à compteur en utilisant la commande `seq` avec un `for` ou bien en utilisant la syntaxe `for (( ... ))` héritée du langage C.

Solutions des exercices

Solution de l'exercice 4.4.1 page 238:

```
#!/bin/bash
DEB=${1:-1}
FIN=${2:-10}
i=$DEB
while [ $i -le $FIN ] ; do
    echo $i
    i=$(( $i + 1 ))
done
```

Solution de l'exercice 4.4.2 page 244:

```
#!/bin/bash
HAUTEUR=${1:-4}
LIGNE="O"
for ((i=1; i<=$HAUTEUR; i++)); do
    echo $LIGNE
    LIGNE="O$LIGNE"
done
```

Activité 4.5

Structures de routines

1 Introduction

Que ce soit en mode interactif ou en mode script, certaines actions sont à faire de manière répétitive et espacée dans le temps. Nous allons voir comment regrouper des commandes pour former comme des petits scripts à l'intérieur d'un script mais aussi à l'intérieur du shell. La notion de fonction sert dans le premier cas à réutiliser un ensemble de commandes. C'est une bonne pratique car elle évite d'avoir du code de script trop long et difficile à lire et à maintenir. Dans le second cas, elle offre le moyen de composer ses propres commandes pour interagir en ligne de commande. En mode interactif, le Bash offre aussi la notion d'alias pour raccourcir la saisie des commandes les plus courantes.

Nous allons dans un premier temps présenter les éléments d'une fonction pour son utilisation dans un script. Nous terminerons cette activité en présentant les moyens de créer ses propres commandes pour que l'utilisateur puisse adapter ses commandes courantes à ses habitudes et à son activité.

2 Notion de fonction

Dans ce paragraphe, nous allons présenter la notion de fonction et voir comment organiser le code de nos scripts. La fonction est une chose assez classique quand on écrit des programmes. Ça permet de factoriser dans une zone unique les traitements qu'on utilise à plusieurs reprises dans le code. On va donc voir comment :

- déclarer et utiliser une fonction dans un fichier script,
- passer des arguments à une fonction,
- définir des variables locales au code d'une fonction et gérer correctement la valeur que doit retourner une fonction.

2.1 Définir une fonction

Pour définir une fonction, on doit procéder ainsi :

- On doit se placer sur une nouvelle ligne du fichier source et écrire le nom que l'on souhaite donner à la fonction.
- Ce nom doit être suivi d'un couple de parenthèses ouvrant/fermant. C'est cette marque qui indique au système qu'on est en train de définir une fonction.
- On ouvre ensuite un bloc d'accolades, qui va contenir la suite de commandes qui expriment le traitement réalisé par la fonction. C'est le corps de la fonction.

```
nom_fonction() {
  commande1
```

```
    commande2
    ...
  }
```

2.2 Invoquer une fonction

Quand on définit une fonction avec la syntaxe ci-dessus, la fonction n'existe que pour le fichier script dans lequel elle a été définie. De plus, la fonction doit être déclarée avant d'en faire l'appel. On prendra donc l'habitude de déclarer systématiquement toutes les fonctions au début des fichiers scripts.

Pour utiliser la fonction, il suffit simplement de taper son nom, comme si c'était une commande shell classique. Ainsi, quand on définit une fonction dans un script c'est un peu comme si on créait une nouvelle commande shell propre à ce script. Voyons cela dans un petit exemple illustratif. On va commencer par créer un fichier script de nom `hello.sh` :

```
$ vim hello.sh
```

On va définir une fonction au début de ce fichier. Par exemple une fonction de nom `say_hello` qui va simplement afficher le texte : "Hello World !!!". On va ensuite déclencher un appel de cette fonction. Il suffit pour cela de taper son nom. Et on va même faire un deuxième appel de cette fonction juste en dessous. Et pour que les choses soient plus visibles, des commentaires sont ajoutés pour bien montrer où se trouve la définition de la fonction, et où se trouvent les appels de celle-ci.

```
#!/bin/bash
#définition de la fonction
say_hello() {
  echo "Hello World !!!"
}
#utilisation de la fonction
say_hello
say_hello
```

On enregistre. On fixe le droit d'exécution, et on lance le script :

```
$ chmod +x hello.sh
$ ./hello.sh
Hello World !!!
Hello World !!!
$
```

On a bien l'affichage attendu.

Il est important de noter que l'exécution de la fonction s'effectue dans le même contexte que l'appelant, à la différence d'un script qui s'exécute par un sous-shell et qui possède son propre contexte. On peut voir la fonction comme équivalent au regroupement de commandes noté entre accolades auquel un nom a été associé pour l'identifier et l'appeler. Pour illustrer le point commun entre une fonction et le regroupement de commandes, prenons l'exemple du test du contenu d'une variable qui est incrémentée et affichée si elle vaut la valeur 10. Avec le regroupement de commandes, nous écrivons un enchaînement conditionnel de la manière suivante :

```
$ [ "$a" -eq 10 ] && { ((a++)) ;  echo $a ;  }
```

La variable `a` est définie dans le contexte du shell, si elle vaut 10, elle passera à 11. En utilisant une fonction, nous devons la définir préalablement à son appel. Dans ce cas, l'exemple devient comme indiqué ci-dessous et nous aurons le même comportement.

```
$ fun() {
> ((a++)) ;  echo $a ;
> }
$ [ "$a" -eq 10 ] && fun
```

Maintenant si nous avons recours au script `fun.sh` ci-dessous :

```
#!/bin/bash
((a++))
echo $a
```

le script s'exécute dans un autre contexte, dans lequel la variable appelée `a` prend la valeur 0 (c'est en fait une autre variable mais avec le même nom). C'est ce que nous pouvons en déduire avec l'exécution ci-dessous :

```
$ a=10
$ [ "$a" -eq 10 ] && fun.sh
1
```

2.3 Les arguments d'une fonction

Comme on vient de le voir, la syntaxe d'appel d'une fonction est identique à celle d'une commande classique. Cela se vérifie aussi pour le cas où on doit passer des arguments à la fonction. Sur la commande d'appel, ces arguments doivent être placés les uns après les autres juste après le nom de la fonction :

```
nom_fonction   arg1  arg2  ...  argN
```

On utilise simplement une ou plusieurs espaces pour les séparer. Les arguments de la fonction sont placés dans des variables positionnelles. Il s'agit en effet des mêmes variables que l'on a déjà utilisées pour exploiter les arguments des scripts. Le corps de la fonction définit et utilise des paramètres notés sous la forme de variables positionnelles. On retrouve donc les variables :

- $1, $2, $3, ... qui vont contenir les arguments passés à l'appel de la fonction.
- $# qui aura pour valeur le nombre d'arguments transmis au moment de l'appel.
- $@ et $* qui contiendront l'ensemble des arguments, le premier dans une liste et le second dans une chaîne de caractères.

C'est le fait de se trouver dans le corps d'une fonction qui va donner cette signification à ces variables. En dehors du code de la fonction, elles retrouvent leur rôle précédent qui est celui lié aux arguments du script.

À titre d'illustration, nous allons modifier notre script `hello.sh` de sorte que la fonction `say_hello` utilise le premier argument qu'on lui transmet dans la phrase qu'elle doit afficher, à la place du mot `World` écrit en dur dans son code. On va aussi passer un argument différent pour chacun des appels placés à la fin du script.

```
#!/bin/bash
#definition de la fonction
say_hello() {
  echo "Hello $1 !!!"
}
```

```
#utilisation de la fonction
say_hello Me
say_hello You
```

On enregistre, et on lance l'exécution.

```
$ ./hello.sh
Hello Me !!!
Hello You !!!
$
```

Voilà qui devrait rendre notre fonction `say_hello` un peu plus utile !

2.4 Les variables locales à une fonction

Quand on manipule une variable dans un script, les changements subis par cette variable valent pour l'ensemble du script. Et c'est aussi le cas pour les variables utilisées dans le code d'une fonction. Aussi, on dit que les variables sont `globales` au script, ou encore que la `portée` d'une variable est globale.

Il est cependant possible de limiter la portée d'une variable dont l'usage ne concerne que l'espace du corps d'une fonction. Cela permet d'éviter d'entrer en conflit avec les autres traitements du script. Car quand le code source d'un script est long, le risque d'utiliser le même nom de variable à deux endroits différents, sans s'en rendre compte, est grand. Pour obtenir cette limitation, il suffit d'utiliser la commande `local`. Cette commande doit se placer au début du code de la fonction, et elle prend en argument le nom de la variable concernée avec éventuellement la valeur de départ de celle-ci. :

```
  local nom_variable
```

ou

```
  local nom_variable=valeur
```

Une fois marquée comme locale, les manipulations réalisées sur la variable dans la fonction ne vont pas avoir d'incidence sur le reste du code script, et ce même si on utilise une variable de même nom ailleurs. En tant que variable locale à la fonction, elle sera toujours considérée comme une nouvelle variable propre à l'exécution de la fonction.

Exercice 4.5.1: Dans un script de nom `fullpyramide.sh` écrire une fonction de nom `buildline`. Cette fonction doit prendre en argument deux nombres A et B afin d'afficher une ligne composée de A caractères espace, suivis de (2*B+1) caractères O. Dans la suite du script utiliser cette fonction pour produire l'affichage d'une véritable pyramide (2 pentes) dont la hauteur sera indiquée en argument du script, comme dans l'exemple ci-dessous :

```
$ fullpyramide.sh 4
    O
   OOO
  OOOOO
 OOOOOOO
$
```

Solution page 257

2.5 Retour des fonctions

L'autre élément qui reste à voir au sujet des fonctions est son code retour. À ce niveau, une fonction shell suit le même principe qu'une commande : son code de retour est une valeur numérique (0 à 255) dont l'objectif est d'indiquer si tout s'est bien passé. Il a souvent la sémantique vrai ou faux.

Pour une fonction, les choses se passent comme pour le code retour d'un script, tel que nous l'avons étudié dans l'activité 4.1. Ce code sera le code retourné par la dernière commande exécutée dans le corps de la fonction, ou celui de la commande `return` qui peut être utilisée à tout moment pour sortir de la fonction.

La commande `return` joue le même rôle que la commande `exit` du script, elle prend en argument la valeur du code à renvoyer, et si cet argument n'est pas précisé sa valeur par défaut sera 0. Du côté de l'appelant de la fonction, le code retour est récupéré en consultant le contenu de la variable `$?` juste après l'appel.

Mettons en pratique l'usage de ce code retour en créant un script de nom `afficheMax.sh` :

```
$ vi afficheMax.sh
```

Dans ce script, on déclare une fonction de nom `max` qui va retourner la plus grande des deux valeurs qu'il reçoit en argument. On utilisera donc une structure conditionnelle `if` pour comparer les deux premiers paramètres, afin de renvoyer avec `return` la valeur du plus grand.

Les valeurs à comparer seront elles-mêmes fournies au script au travers des arguments qu'il recevra à son lancement. On va donc s'assurer dans le code que le script est bien lancé avec deux valeurs en arguments avant de déclencher l'appel de la fonction. Puis on termine en récupérant la valeur retournée par la fonction pour afficher la réponse dans la console.

```
#!/bin/bash
max() {
  if [ $1 -gt  $2 ]; then
    return $1
  else
    return $2
  fi
}
IS_OK=${2:?"Vous devez fournir 2 valeurs"}
max $1 $2
echo $?
```

On enregistre ce fichier, on fixe le droit d'exécution et on peut faire des essais avec différents jeux d'arguments d'appel :

```
$ chmod +x  afficheMax.sh
$ ./afficheMax.sh  16 157
157
$ ./afficheMax.sh  58 22
58
$
```

On peut constater que tout fonctionne correctement. Mais si on pousse un peu les tests et qu'on utilise des valeurs supérieures à 255, on va remarquer que ça ne fonctionne plus du tout :

```
$ ./afficheMax.sh  16 490
234
```

```
$ ./afficheMax.sh  580 127
68
$
```

Cela vient du fait que dans l'écriture de notre fonction on fournit la réponse du maximum via la valeur de retour de la fonction. Or comme nous l'avons vu lorsque nous avons examiné les valeurs de retour d'une commande, cette valeur est limitée à un code compris entre 0 et 255 !

✍ **Challenge C45Q1**

Ploumploum

Si on souhaite créer une fonction qui fournit un résultat autre qu'un nombre entre 0 et 255, par exemple un entier quelconque ou une chaîne de caractères, alors il faut utiliser une technique de contournement. Il existe 2 stratégies pour y parvenir :

- La première consiste à utiliser une variable globale au script. À la fin de son traitement la fonction devra fixer la valeur produite dans cette variable. Le code qui a invoqué la fonction n'aura alors qu'à consulter la valeur de cette variable juste après la commande d'appel.
- La seconde stratégie consiste à envoyer le résultat de la fonction sur la sortie standard, avec la commande `echo`. On récupère ensuite cette valeur en utilisant la syntaxe de substitution de commande : `$( ... )`.

Nous allons améliorer notre script `afficheMax.sh` avec cette seconde technique :

```
$ vi afficheMax.sh
```

Au lieu de retourner la valeur maximum dans la fonction `max`, nous allons cette fois l'afficher avec la commande `echo`. On utilisera ensuite cette fonction en capturant l'affichage qu'elle produit pour placer cette valeur dans une variable de nom `REP`. On terminera le script en affichant la valeur de `REP` à l'écran.

```
#!/bin/bash
max() {
  if [ $1 -gt  $2 ]; then
    echo $1
  else
    echo $2
  fi
}
IS_OK=${2:?"Vous devez fournir 2 valeurs"}
REP=$(max $1 $2)
echo $REP
```

On enregistre et on reproduit les tests problématiques :

```
$ ./afficheMax.sh  16 490
490
$ ./afficheMax.sh  580 127
580
$
```

Cette fois tout se passe bien !

Exercice 4.5.2: En utilisant la même fonction `max` ci-dessus (elle devra être ré-écrite dans le nouveau script), écrire un script de nom `max3.sh` qui affiche la plus grande valeur parmi les 3 arguments fournis dans sa commande de lancement. Si le script ne reçoit pas 3 arguments, un message d'erreur devra être affiché.

Solution page 257

✍ **Challenge C45Q2**

Alice change de braquet

3 Routines en mode interactif

Nous avons vu jusqu'ici les scripts et les fonctions. Les scripts sont des programmes autonomes qui permettent d'effectuer des tâches précises. Nous avons vu aussi que les fonctions sont des routines utilisées pour factoriser un traitement. Jusqu'ici, nous avons défini les fonctions dans les fichiers contenant les scripts. Nous allons étendre ces 2 niveaux des routines avec la notion d'alias. Nous allons revenir ensuite sur les trois niveaux de besoin afin de savoir quand utiliser les scripts, les fonctions ou les alias.

3.1 Les alias

Les alias sont des moyens de raccourcir la saisie de commande. Un alias est souvent employé pour une commande dont les options utilisées sont toujours les mêmes. Vous êtes maintenant à l'aise devant votre ligne de commande et vous commencez à avoir vos habitudes. Nous avons vu dans l'activité 3.1 comment créer des variables, ici nous allons créer un raccourci pour des commandes, des options ou des arguments. Pour cela nous utilisons la commande `alias`. Les alias peuvent être définis en mode interactif mais le plus souvent ils sont définis dans un fichier tel que `.bashrc` ou `.profile`.

La commande `alias`, sans argument liste tous les alias actuellement définis dans le shell. Dans l'exemple ci-dessous, il y a un alias de défini.

```
$ alias
alias ll='ls -al'
```

La définition d'un alias suit la syntaxe suivante :

```
alias nomAlias="commande à exécuter"
```

Cette commande crée l'alias `nomAlias` pour être utilisé à la place de la `commande à exécuter`. Par exemple, l'alias précédent effectue la commande `ls -al` en saisissant `ll`. Dans la Weblinux, il est défini dans le fichier `.bashrc`.

```
$ cat .bashrc
[...]
alias ll='ls -al'
[...]
$ ll
[...]
-rw-r--r--    1 alice    user           170 Mar  3 16:15 .bashrc
-rw-r--r--    1 alice    user           625 Mar  3 16:15 .profile
drwxr-xr-x    1 alice    user             0 Mar  3 16:15 Books
```

```
drwxr-xr-x    1 alice    user            0 Mar  3 16:15 Documents
drwxr-xr-x    1 alice    user            0 Mar  3 16:15 Movies
drwxr-xr-x    1 alice    user            0 Mar  3 16:15 Music
[...]
```

La commande `unalias` supprime un alias existant.

```
$ unalias ll
$ alias
```

Dans l'ensemble des commandes précédentes, nous avons supprimé l'alias `ll`. En règle générale les alias s'utilisent pour spécifier les arguments ou les options d'une commande que l'on utilise fréquemment. Par exemple, pour demander une confirmation à chaque suppression de fichier, on peut utiliser l'alias suivant :

```
$ alias rm='rm -i'
```

Nous voyons dans ce cas que la commande `rm` est un alias de la commande `rm -i`. Dans l'exemple, la commande `rm` est redéfinie. Voici un exemple de l'utilisation de la commande `rm` avec et sans son alias.

```
$ touch test
$ rm test
$ alias rm='rm -i'
$ touch test
$ rm test
rm: remove 'test'? y
```

Nous voyons que pendant la deuxième utilisation de la commande `rm` une confirmation est demandée. Pour utiliser la commande `rm` sans utiliser l'alias, il faut ajouter une inhibition caractère devant le nom de la commande de la manière suivante : `\commande`. Dans notre exemple, l'appel de la commande `rm` directement sans passer par l'alias s'effectue ainsi :

```
$ touch test
$ \rm test
```

Les alias sont très importants car ils permettent de gagner en efficacité et permettent d'adapter réellement l'utilisation de la ligne de commande à l'utilisateur. Cependant il faut comprendre que les alias sont une substitution. Le nom de l'alias est remplacé par la commande ou plus généralement par la chaîne de caractères associée au nom d'alias. L'alias est une opération syntaxique effectuée par le Bash. L'alias n'existe pas en phase d'exécution de la commande. Par conséquent, il n'est pas possible de passer des arguments à utiliser dans la chaîne de caractères associée à l'alias. Il n'y a pas d'appel d'alias comme pour une fonction. C'est là la principale limitation dans l'usage de l'alias.

3.2 Les fonctions en mode interactif

Nous avons vu dans le paragraphe précédent l'utilité des alias et leurs utilisations. Malgré leur efficacité, les alias sont toujours des commandes assez simples et ne permettent pas des traitements complexes. Notamment, ils ne tolèrent pas les arguments. Dans ce cas, il faut avoir recours à une fonction. Il est possible de définir des fonctions dans les fichiers `.bashrc` ou `.profile` qui permettent d'avoir des traitements plus complexes. Les fonctions définies ainsi sont des commandes situées entre la simplicité d'un alias et la complexité d'un script. Prenons un exemple d'utilisation d'une fonction que nous définissons dans le fichier `.bashrc`.

```
$ cat ~/.bashrc
[...]
mkd() {
    mkdir -p "$1" && cd "$1"
}
[...]
$ source ~/.bashrc
```

Ici, nous avons défini une fonction du nom de `mkd`. Cette fonction crée un répertoire dont le nom est donné en argument et se positionne dans ce répertoire si la création s'est bien déroulée. La commande `source .bashrc` permet de "recharger" le fichier `.bashrc` pour prendre en compte les modifications dans le shell courant. Voici un exemple d'utilisation de la fonction.

```
$ mkd /tmp/t/t/t/t/t/t/t/t/t/t
$ pwd
/tmp/t/t/t/t/t/t/t/t/t/t
```

Notons ici que l'option `-p` de la commande `mkdir` permet de créer les répertoires intermédiaires s'ils n'existent pas. Il est possible d'utiliser dans une fonction des structures conditionnelles et des structures itératives comme on le ferait dans un script.

On pourrait se demander la différence entre un script et une fonction. Il en existe plusieurs mais nous n'allons pas entrer dans les détails de ces différences ici. Nous pouvons simplement dire qu'une fonction, comme définie ici, ne peut pas s'utiliser dans un script alors qu'un script pourrait très bien être utilisé dans un autre script.

Une bonne pratique pourrait simplement consister à commencer par créer un alias, si le traitement est trop complexe pour un alias, faire une fonction et si une fonction dépasse une quinzaine de lignes, faire un script.

Une autre bonne pratique est de séparer les fichiers contenant les alias et les fonctions en créant des fichiers `.alias` et `.func` et de rajouter dans le fichier `.bashrc` les lignes suivantes :

```
$ cat .bashrc
[...]
source .alias
source .func
[...]
```

4 Conclusion

Au terme de cette activité vous savez maintenant l'essentiel sur les routines du Bash et en particulier vous avez pu voir en détail la notion de fonction. C'est-à-dire :

- définir et invoquer une fonction
- passer des arguments à une fonction
- exploiter ces arguments sous la forme de variables positionnles dans le corps de la fonction
- spécifier des variables locales au corps d'une fonction
- les limitations concernant le code retour d'une fonction
- et, comment contourner ces limites pour retourner une valeur quelconque.

En mode script, une fonction sert à factoriser des traitements internes au script. Vous avez pu aussi découvrir les différentes routines utilisables dans un shell interactif. Les routines servent ici à rendre l'interaction en ligne de commande plus efficace et plus simple en adaptant les commandes aux actions

courantes de l'utilisateur. L'alias est une simple substitution opérée par le shell. La fonction est un appel dans le contexte courant de l'appelant et se limite à quelques commandes. Le script est exécuté dans un sous-shell et dans un contexte différent de l'appelant.

Solutions des exercices

Solution de l'exercice 4.5.1 page 250:

```
#!/bin/bash
buildline() { #$1 espaces, suivies de (2*$2 + 1) symboles O
    local i #pour ne pas que la variable i des boucles for
            #ci-dessous ne perturbe celle d'une autre boucle
    local LINE=""
    #les $1 caractères espace
    for ((i=1; i<=$1; i++)); do
        LINE="$LINE "
    done
    #les (2*$2 + 1) caractères O
    LINE="${LINE}O"
    for ((i=1; i<=$2; i++)); do
        LINE="${LINE}OO"
    done
    echo "$LINE"
}
HAUTEUR=${1:?"Vous devez indiquer la hauteur de la pyramide"}
i=0
while [ $i -lt $HAUTEUR ] ; do
    buildline $(( $HAUTEUR - $i )) $i
    i=$(( $i + 1 ))
done
```

Solution de l'exercice 4.5.2 page 253:

```
#!/bin/bash
max() {
    if [ $1 -gt $2 ]; then
        echo $1
    else
        echo $2
    fi
}
IS_OK=${3:?"Vous devez fournir 3 valeurs"}
REP=$(max $1 $2)
REP=$(max $REP $3)
echo $REP
```

Conclusion

C'est le moment de faire le bilan de cette séquence. Nous avons vu progressivement chacun des aspects permettant d'exploiter le Bash comme un véritable langage de programmation.

Avec d'abord un tour d'horizon complet des éléments composant un script shell. Nous avons ensuite appris à écrire des expressions de test, afin de les utiliser pour réaliser des structures conditionnelles. Ensuite, nous avons eu l'occasion de découvrir les boucles, qui permettent d'écrire des opérations répétitives en quelques lignes. Enfin, nous avons appris qu'on peut organiser plus efficacement le code des scripts en écrivant des fonctions.

Pour conclure, nous vous proposons d'énumérer les bonnes pratiques dans l'écriture d'un script shell. En premier lieu, la règle d'or dans l'écriture d'un script shell est la lisibilité. Pour cela il faut favoriser les structures simples et commenter les passages complexes. Ensuite, les principaux conseils sont de :

- soigner la présentation par l'indentation, les commentaires et décrire les objectifs du script,
- favoriser les interfaces du script en ligne de commande (et non par interactivité),
- proposer une aide interne par l'option `-h`,
- nommer les variables avec un nom descriptif,
- placer les valeurs constantes dans des variables en début de script ; n'utiliser aucune constante littérale dans le code,
- déclarer les variables dans les fonctions avec la commande `local`,
- activer les éléments de prévention aux erreurs : `set -u, noclobber`
- vérifier le code retour des commandes exécutées,
- afficher les messages d'erreur sur la sortie d'erreur en détaillant l'erreur et la cause,
- utiliser les fonctions et les ranger dans des bibliothèques ; indiquer en commentaire les paramètres de la fonction,
- réaliser des scripts intégrés c'est-à-dire avec les codes spécifiques des différents outils dans le script lui-même.

Après ces bonnes pratiques, il vous faut éviter les erreurs courantes, à savoir :

- oublier les autorisations d'exécution,
- oublier le shebang pour un fichier exécutable,
- ne pas ajouter le répertoire courant dans le PATH,
- nommer un script comme une commande interne par exemple : test,
- oublier les guillemets lors des affectations et des tests,
- oublier qu'un tube crée un sous-shell,
- accéder à une variable non initialisée, pour détecter cela utiliser `set -u`,
- utiliser une fonction avant sa déclaration,
- confondre chaînes de caractères et contenu de fichiers,
- oublier de placer des espaces autour des crochets, et du caractère `;`

Vous avez maintenant toutes les clés en main pour créer vos propres commandes Bash, et concevoir des routines personnalisées pour automatiser tous vos traitements !

Séquence 5

Errata

Errata

Introduction

En concevant et réalisant ce cours, nous avons essayé de suivre et d'appliquer une méthode pour atteindre un bon niveau de qualité. Malgré ce soin apporté au document compagnon, il reste des coquilles à corriger, des oublis à réparer, des passages à préciser etc. Avec leur regard neuf, les apprenants ont un rôle essentiel dans cette démarche qualité. Leurs remarques et suggestions faites dans les forums seront reçues et traitées avec attention. Au travers de ce document, nous voulons permettre à tout un chacun de voir les évolutions du document compagnon. En effet il nous semble que le numéro de version du document compagnon est une information importante mais non suffisante. Ce chapitre vise à donner ce complément d'information par la description des modifications faites à chaque changement de version.

Version 3.1

Séquence 0

— A 0.4 : Explication de l'usage bouton sauvegarde de la Weblinux

Séquence 1

— A 1.1 : correction d'une coquille p.27 sur la commande externe `/usr/bin/man`
— A 1.1 : présentation de la commande echo, ajout du mot "les" qui manquait dans la première phrase.
— A 1.3 : présentation de `ls -l`, changé le terme date de création en date de modification.
— A 1.4 : Re-formulation du paragraphe 3.4 "Changer les droits par défaut, la commande umask". Le terme "différence" employé pour expliquer comment déterminer les droits à partir de la valeur du masque prêtait à confusion car il s'agit en fait de soustraire les droits du masque dans le sens "enlever". La méthode d'attribution des droits à partir du masque a été détaillée et la section ré-écrite.

Séquence 3

— A 3.1 : Correction d'un souci avec un balisage incorrect.
— A 3.2 : Correction d'un chemin incorrect dans l'exercice 3.2.1.
— A 3.3 : Présentation de l'utilisation de base de la commande sed : suppression d'un "un" superflu dans une phrase, et correction d'une coquille sur le nom de la commande dans la première phrase.
— A3.3 : Suppression d'une accolade superflue dans la solution de l'exercice 3.3.1.
— A 3.4 : Correction d'une coquille : "avons" remplacé par "savons"
— A 3.4 : Correction d'un chemin erroné dans l'exercice 3.4.1.

Version 3.2

Séquence 1

— A1.4 : Paragraphe 3.3, ajout de la phrase « Si le public n'est pas renseigné il vaut par défaut `ugo`. » à propos du public de la commande `chmod`.
— A1.4 : Paragraphe 3.4, correction d'une typo dans l'exemple de création de fichier et d'utilisation de la commande `umask`.
— A1.5 : Paragraphe 2.1, correction de l'option utilisée avec la commande `cat` pour obtenir la numérotation des lignes.

Séquence 2

— A2.2 : Section 3, re-formulation de la description des motifs étendus (extglob) (la version précédente était ambiguë).

Séquence 3

— A3.4 : Suppression de l'exemple de mettre les variables en inhibition partielle dans expr.
— A3.5 : Correction de quelques soucis typographiques dans la description des options de la commande `tar`.

Séquence 4

— A4.1 : Correction de divers petits soucis d'orthographe
— A4.1 : Reformulation de la description de l'effet de la commande `shift` lors de l'exécution du script `decal.sh`.
— A4.1 : Suppression de la dernière phrase de la sous-section 4.1
— A4.1 : Ajout d'un caractère $ manquant dans la définition de la variable pseudo1, et remplacement des variables pseudo2 et PSEUDO1 par pseudo1 dans les exemples concernant la portée des variables
— A4.1 : Correction d'un souci de mise en forme de la syntaxe de consultation des arguments pour des variables positionnelles ayant plus de un chiffre
— A4.1 : Ajout d'un exemple de l'utilisation de la version abrégée de la commande source
— A4.3 : Correction d'un caractère } et d'un caractère ! superflus dans les exemples d'utilisation des structures conditionnelles
— A4.3 : Correction de quelques typos dans l'énoncé et la correction de l'exercice 4.3.3
— A4.4 : Correction d'un souci d'orthographe dans le rappel sur les expressions conditionnelles
— A4.4 : page 234, erreur typo dans `while [ $i -lt $nb ]` et ajout de la substitution variable à la variable ligne `echo $a $ligne`
— A4.4 : page 235, correction du résultat de la commande `./print.sh print.sh` dans lequel figurait un appel à `cat` superflu.

Version 3.3

— A0.4 : Suppression des références à la weblinux avec authentification. Elle n'est pas utilisée dans le cours.
— A0.4 : Ajout d'une précision sur la taille des transferts par le répertoire shared.

Séquence 1

— A1.2 : correction d'un souci orthographique en page 31.
— A1.3 : page 46, suppression de l'allusion à l'option `-d` de la commande `rm`, cette option n'existe pas dans la norme POSIX sur laquelle est fondée le shell Bash.

Séquence 2

— A2.3 : Ajout d'une section "Manipulation de chaînes de caractères".

Séquence 3

— A3.3 : Remplacement de l'option `_iname` par l'option `-name` dans les exemples d'illustration de l'option `-exec` de la commande `find`.

Séquence 4

— A4.1 : Ajout d'une section "Affectation des variables positionnelles" pour l'explication de la commande `set`.
— A4.3 : Ajout d'une sortie par exit sur l'exemple de la page 227 et ajout d'explications sur le début de l'exemple.
— A4.4 : Positionnement et nommage correct des challenges
— A4.4 : section 3.2, ajout d'une précision sur l'utilisation de la commande `read` qui aide à la réalisation du challenge C44Q1.
— A4.4 : section 4.1, ajout d'un caractère $ manquant dans l'exemple d'utilisation de la boucle `for`.

Version 3.4

Séquence 1

— A1.4 : section 3.1, p.54, ajout d'une précision sur le public "groupe" concernant les droits d'un fichier

Séquence 2

— A2.2 : p. 91, ajout du répertoire fac dans le résultat de la commande `echo *f*`
— A2.2 : p. 95, modification du premier exemple qui ne justifiait pas de l'utilisation de la substitution étendue.
— A2.3 : page 102, remplacement du mot préfixe par suffixe dans l'explication du caractère

Séquence 3

— A3.3 : p.170, ajout du caractère . manquant dans l'expression régulière d'un appel à `sed`

Séquence 4

— A4.1 : p. 207, correction d'une typo dans l'appel du script `nbArg`
— A4.3 : reformulation de l'énoncé de l'exercice 4.3.2 pour le mettre en conformité avec la solution proposée.
— A4.4 : correction de deux typo dans la présentation des instructions `break` et `continue`.

— A4.4 : p. 238, ajout du chemin ./ devant l'appel au script `print.sh`.