

Initiation au Shell

Z. Hamrouni

Vous débutez votre formation d'ingénieur, et vous serez amenés à travailler régulièrement sur les ordinateurs de l'Ecole, et vous devez être opérationnel rapidement, dès les premiers TP. Ce document a pour objectif de vous fournir une présentation ciblée des principaux outils nécessaires à une prise en main rapide de votre environnement de travail informatique.

1- Le réseau informatique à l'N7

L'N7 met à votre disposition un réseau de plusieurs centaines d'ordinateurs organisés en réseau, de manière à vous permettre de travailler sur n'importe quel ordinateur, dans n'importe quelle salle de TP, et même depuis chez vous comme nous le verrons plus loin.

Ces ordinateurs sont répartis dans une vingtaine de salles de TP, et fonctionnent en grande majorité sous **le système Linux**.

Pour y accéder, vous disposez d'un compte (login et mot de passe fournis lors de l'inscription). Vos données (fichiers, programmes, documents, etc.) sont stockées sur un serveur et sont accessibles depuis tout ordinateur du réseau. Elles sont régulièrement sauvegardées, de manière à éviter toute éventuelle perte due à une panne. Votre **espace de données est donc bien sécurisé** : il est à l'abri de toute panne matérielle, et vous êtes seul à pouvoir y accéder si vous ne divulguez pas votre login et votre mot de passe. Il est donc essentiel de choisir un mot de passe robuste (une dizaine de caractères combinant minuscules, majuscules, chiffres et caractères spéciaux), et de ne fournir sous aucun prétexte ce mot de passe à une tierce personne. Vous êtes aussi responsable de toutes les activités réalisées depuis votre compte (téléchargements, exécution de programmes, etc.) comme le stipule la charte informatique.

Mais cet **espace est limitée**, et vous devez le gérer de façon efficace : ne pas y mettre n'importe quoi, le nettoyer régulièrement des fichiers volumineux, bien l'organiser pour pouvoir retrouver vos données facilement et rapidement. Car une saturation de votre espace de données bloque toute activité sur votre compte, et peut être très pénalisante si elle a lieu en milieu d'une séance de TP.

Vos activités sur les ordinateurs de l'N7

Pour les besoins de votre formation, vous serez amenés à créer des programmes informatiques dans des langages spécifiques (Python, C, Ada, java, ou autre), à en générer une version exécutable, à les exécuter pour tester leur fonctionnement et leur robustesse, à générer des rapports, à leur donner des noms significatifs, à les stocker à des endroits bien choisis pour pouvoir les retrouver facilement. Le nombre de ces fichiers va grossir au fil du temps, et vous devez donc bien organiser votre espace pour les ranger de façon efficace et pouvoir les retrouver facilement et rapidement.

Vous devrez donc savoir :

- créer des répertoires ou dossiers (une pièce par année, des armoires pour les différentes matières, des étagères pour les activités (TP, projet, etc.)
- les déplacer si nécessaire, les vider ou détruire lorsqu'ils deviennent inutiles
- créer des fichiers en utilisant et en maîtrisant un éditeur adapté : programme informatique, rapport, ...

- les déplacer si nécessaire, ou les détruire lorsqu'ils deviennent inutiles
- éventuellement les compiler pour en générer un exécutable
- exécuter un programme, tester le fonctionnement, l'arrêter s'il boucle indéfiniment
- supprimer l'exécutable une fois les tests terminés, car il est volumineux et risque de contribuer à saturer votre espace, d'autant plus qu'il peut être régénéré à tout moment à partir du code initial
- vérifier le taux d'occupation de votre espace de données
- retrouver les fichiers les plus volumineux pour libérer de la place si nécessaire, en les supprimant ou en les déplaçant sur un autre support (clé usb, votre ordinateur personnel).

Il est probable que vous sachiez déjà faire une bonne partie de cela. Mais :

- vous avez très probablement travaillé sous « Windows », et/ou avec des interfaces graphiques ou dans des logiciels intégrés différents de ceux utilisés à l'N7
- les interfaces graphiques sont plus pratiques seulement pour les opérations les plus courantes et les plus simples
- en travaillant à distance sur les ordinateurs de l'N7, vous n'aurez pas toujours un accès facile aux outils graphiques (systèmes incompatibles, outils lourds, faible débit internet, etc.)
- certaines tâches doivent être automatisées, car répétitives ou étalées dans de temps, et sont beaucoup plus faciles à réaliser avec une commande adaptée ou un petit script.

Ce document, et le jeu de piste associé (réalisé en TP), ont pour objectif de vous conduire à une maîtrise des outils de base nécessaires à vos activités sur les ordinateurs de l'N7, aussi bien en distanciel qu'en présentiel.

2- Le système Linux et le shell

Une grande partie des ordinateurs de l'N7 tournent avec le système Linux, et la grande majorité de vos activités seront réalisées sur ces machines.

Linux, basé sur Unix, est un système d'exploitation libre, efficace, sûr, largement répandu, et dont vous pouvez trouver l'histoire originale dans différents tutoriaux ou documents internet.

Mais qu'est-ce un système d'exploitation ?

Tout outil numérique avancé (ordinateur, tablette smartphone, etc.) a besoin d'un logiciel qui sert d'intermédiaire entre les ressources matérielles (processeur, mémoire, disques de stockage, entrées/sorties, etc.) et les applications qui l'utilisent. Ce gros logiciel s'appelle système d'exploitation.

Il permet de gérer, d'allouer et de partager les ressources matérielles :

- vous n'avez pas besoin de manipuler directement le disque dur de l'ordinateur pour y stocker de nouveaux fichiers, en déplacer ou en détruire. C'est le système d'exploitation qui offre les fonctions nécessaires pour effectuer ses opérations

- un programme exécutable n’a pas besoin de manipuler directement le processeur et la mémoire pour être exécuté. C’est le système d’exploitation qui alloue au programme exécutable du temps processeur, de la mémoire, l’accès aux entrées/sorties, etc. Et c’est le système d’exploitation qui gère l’allocation et le partage de toutes les ressources entre les différents programmes ou applications qui sont exécutés en pseudo-parallélisme.

Le système d’exploitation permet de masquer les détails et les particularités de mise en œuvre : une application n’a pas besoin de connaître le type de disque ni son mode de fonctionnement : le système lui offre des fonctions d’ouverture, de lecture et d’écriture de fichiers.

Il permet de fournir des abstractions : pour Unix/Linux, les dispositifs d’entrées/sorties (claviers, écran, fichiers, tubes de communication, etc.) sont identifiés par un descripteur (numéro), et sont accessibles par les mêmes fonctions d’ouverture, lecture, écriture et fermeture. Ce qui permet à une application d’accéder de la même manière à tout dispositif d’entrées/sorties. Nous verrons plus loin comment un programme conçu pour lire des entrées au clavier peut être exécuté en lisant ses entrées dans un fichier sans aucune modification du code.

Linux, comme la majorité des systèmes, offre deux interfaces :

- Interface “programmative”, ou API (*Application Programming Interface*)
 - utilisable dans des programmes (applications)
 - composée d’un ensemble d’appels systèmes (appels de procédures avec paramètres)
- Interface utilisateur, ou interface de commande
 - utilisable par un usager humain sous forme textuelle
 - composée d’un ensemble de commandes, par exemple :
 - ls : liste le contenu d’un répertoire ou dossier
 - cp fichier1 fichier2 : copie fichier1 dans fichier2

LE SHELL : interface utilisateur

Il s’agit d’un interpréteur de commandes (un logiciel qui lit des commandes au clavier, les interprète et les exécute). Il est exécuté automatiquement à l’ouverture d’un nouveau terminal (ou console). Il permet à l’utilisateur d’accéder à un grand nombre de fonctionnalités : lister le contenu d’un répertoire, copier, déplacer ou supprimer des fichiers et des dossiers, se déplacer dans l’arborescence des données, lister et gérer les processus, etc.

Le shell exécute des commandes internes implantées dans le shell lui-même, ou tout programme exécutable externe au shell (vos propres programmes par exemple).

A l’origine de Unix, le shell de base était sh, qui donna naissance à de nombreuses variantes, dont csh, étendu en tcsh, ou ksh. Mais aujourd’hui bash (born/bourne again shell) est le shell le plus répandu, et qui offre de nombreuses facilités. C’est bash que vous utiliserez par défaut à l’N7, mais vous pourrez en utiliser un autre si vous le souhaitez.

Le shell est un programme qui, dans une boucle quasi-infinie (voir algorithme dessous), affiche un invite (prompt, par exemple login@machine: dossier_courant\$), attend l’entrée d’une commande par l’utilisateur, l’exécute si elle correspond à une fonction interne, ou à un exécutable externe :

Répéter

```
    afficher le prompt (invite)
    lire une commande
    analyser la commande // options, extensions, commande interne ou externe, ...
    si la commande existe
        exécuter la commande
    sinon
        afficher un message d'erreur
    fin
```

jusqu'à ARRET

3- Quelques commandes de base

Toute commande prend la forme suivante (une suite de mots séparés par des espaces, ou blancs) :
commande [options] [arguments]

commande est le nom de la commande à exécuter (abréviation : ls pour list)

des éventuelles options, commençant souvent par – (ls –l : lister au format long)

des éventuels arguments : ls, ls fichier1 fichier2, ls *.c

Pour connaître le fonctionnement complet d'une commande, il existe un manuel basique accessible par la commande « man » : man ls par exemple ; mais il existe des versions mieux lisibles et plus conviviales sur internet.

Voici ci-dessous quelques commandes de base :

pwd : print working directory : affiche le chemin du répertoire courant

mkdir nom_dossier (make directory) : crée un répertoire

cd nom_dossier (change directory) : se place dans un dossier

ls, ls -a, ls -l, ls -t, ls -lt : liste le contenu d'un dossier sous différentes formes

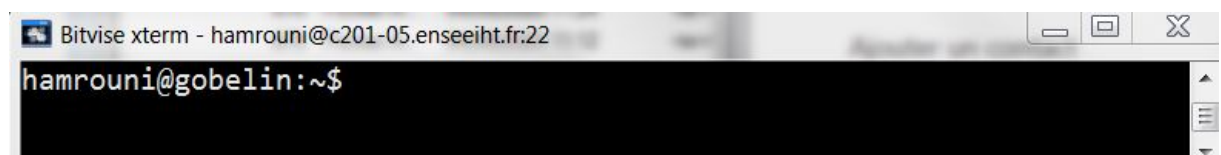
cp fichier1 fichier2 (ou dossier) : copie fichier1 dans fichier2 (dans dossier)

mv fichier1 fichier2 (move) : renomme fichier1 en fichier2, ou déplace dans un dossier

rm fichier (remove) : supprime un fichier / rmdir repertoire : supprime un répertoire vide

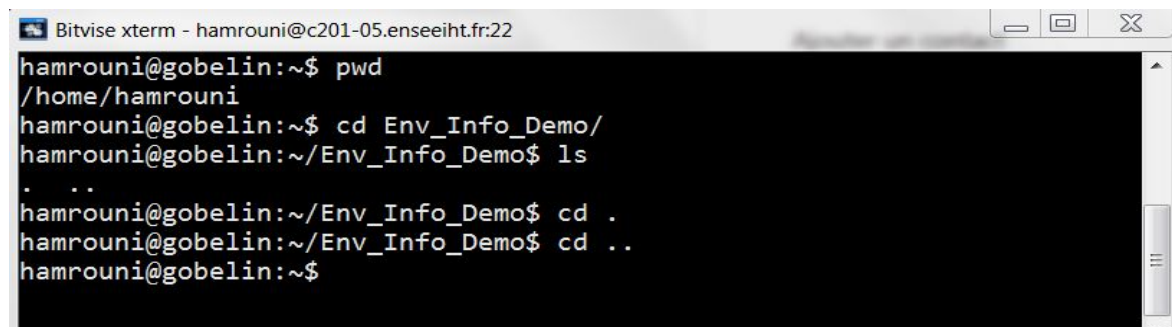
Terminal : c'est une fenêtre basique dans laquelle est lancé un shell (bash dans notre cas), qui va lire des commandes au clavier, les exécuter et afficher les résultats. On peut en ouvrir un certain nombre via le menu de l'environnement graphique, ou l'outil de connexion à distance :

- on voit le prompt (invite) affiché sous la forme login@nom_machine: dossier_courant\$. Ici, on est à la racine (~) du compte « hamrouni » sur la machine « gobelin ».



On exécute les commandes suivantes :

- la commande ***pwd*** affiche le nom du répertoire courant `/home/hamrouni` : c'est le répertoire racine du compte « hamrouni », appelé aussi « home », et accessible par le caractère `~` ou la variable d'environnement ***\$HOME*** (voir plus loin). Vos données sont organisées sous la forme d'une arborescence de répertoire (ou dossiers) :
 - La racine est appelé « HOME »
 - Chaque feuille représente un répertoire, un fichier, ou un lien
 - L'accès à chaque feuille se fait en indiquant son chemin, soit de façon absolue à partir de la racine, soit de façon relative à partir du répertoire courant (répertoire ou on se trouve à l'instant présent)
 - Le chemin est constitué de la suite des noms des répertoires à traverser séparés par `/`
- la commande ***cd Env_Info_Demo*** permet de se placer dans le répertoire `Env_Info_Demo` qui se trouve dans le répertoire courant. Remarquez le changement du prompt.
- La commande ***ls*** affiche le contenu du répertoire courant : on ne voit que « `.` » et « `..` » car le répertoire est vide. Le « `.` » désigne le répertoire courant et le « `..` » le répertoire parent.
- ***cd .*** : on reste dans le répertoire courant
- ***cd ..*** : on remonte dans le répertoire parent



```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~$ pwd
/home/hamrouni
hamrouni@gobelin:~$ cd Env_Info_Demo/
hamrouni@gobelin:~/Env_Info_Demo$ ls
.  ..
hamrouni@gobelin:~/Env_Info_Demo$ cd .
hamrouni@gobelin:~/Env_Info_Demo$ cd ..
hamrouni@gobelin:~$
```

- on revient dans `Env_Info_Demo`,
- la commande ***mkdir Programmes*** : crée un nouveau répertoire « Programmes ». On s'y place avec la commande ***cd Programmes***
- la commande ***cp fichier_source fichier_destnation*** (ou dossier) va :
 - prendre une copie du premier fichier qui s'appelle ici `cal_factorielle.adb` et qui se trouve dans le dossier `App_auto` qui se trouve dans le répertoire `SEC20` qui se trouve à la racine référencée ici par `~` : c'est le chemin qui indique la référence complète de l'objet (fichier ou dossier). Le caractère « `/` » indique le passage d'un étage à un autre. Le chemin est indiqué de façon absolue (en partant de la racine).
 - et l'enregistrer dans le dossier courant « `.` » sous le même nom car on n'en précise pas un (sinon on mettrait `./nouveau_nom` ou `nouveau_nom`)
- la commande ***ls*** montre bien qu'il y a un nouveau fichier « `cal_factorielle.adb` » dans le répertoire courant.

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~$ cd Env_Info_Demo
hamrouni@gobelin:~/Env_Info_Demo$ mkdir Programmes
hamrouni@gobelin:~/Env_Info_Demo$ cd Programmes/
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ cp ~/SEC20/App_auto/cal_factorielle.adb .
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_factorielle.adb
```

- la 2^{ème} commande cp va :
 - prendre une copie du même premier fichier, mais la racine est référencée ici par \$HOME (variable d'environnement prédéfinie)
 - et l'enregistrer sous un nom différent « cal_fact2.adb »
- la 3^{ème} commande cp va :
 - prendre une copie du même premier fichier, dont le chemin est indiqué ici de façon relative par rapport au répertoire courant (on remonte à la racine grâce au ../../)
 - et l'enregistrer sous un nom différent « cal_fact3.adb »

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ cp $HOME/SEC20/App_auto/cal_factorielle.adb cal_fact2.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_fact2.adb  cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ cp ../../SEC20/App_auto/cal_factorielle.adb cal_fact3.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_fact2.adb  cal_fact3.adb  cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

- la commande **ls** peut prendre différentes options : **ls -l** affiche le contenu au format long.
 - Le premier champ indique le type d'objet (d pour directory), suivi des droits d'accès (on y reviendra plus tard)
 - Le 3^{ème} champ indique le propriétaire
 - Le 4^{ème} indique le groupe
 - Le 5^{ème} indique la taille en octets (celle des dossiers n'est pas significative)
 - Les suivants indiquent la date et l'heure de mise à jour

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -l
total 20
drwxr-xr-x 2 hamrouni gea 4096 sept.  4 11:24 .
drwxr-xr-x 3 hamrouni gea 4096 sept.  4 11:08 ..
-rw-r----- 1 hamrouni gea  675 sept.  4 11:13 cal_fact2.adb
-rw-r----- 1 hamrouni gea  675 sept.  4 11:24 cal_fact3.adb
-rw-r----- 1 hamrouni gea  675 sept.  4 11:12 cal_factorielle.adb
```


- En ajoutant l'option « t » (*ls -l -t* ou *ls -lt*) on affiche les objets par ordre chronologique décroissant (les plus récents en premier).

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -lt
total 20
-rw-r----- 1 hamrouni gea 675 sept. 4 11:24 cal_fact3.adb
drwxr-xr-x 2 hamrouni gea 4096 sept. 4 11:24 .
-rw-r----- 1 hamrouni gea 675 sept. 4 11:13 cal_fact2.adb
-rw-r----- 1 hamrouni gea 675 sept. 4 11:12 cal_factorielle.adb
drwxr-xr-x 3 hamrouni gea 4096 sept. 4 11:08 ..
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

- La commande **rm** (remove) supprime un fichier. Ici on détruit « cal_fact3.adb » : à utiliser avec précaution, car rm ne garde pas de copie dans la poubelle
- La commande **mv** (move) déplace un fichier. Ici, elle revient à renommer « cal_fact2.adb » en « cal_fact_nouv.adb »
- La 2^{ème} commande mv déplace « cal_fac_nouv.adb » dans le nouveau dossier « copies » créé avec **mkdir** (« Copies » étant reconnu comme répertoire existant, on garde le même nom de fichier car on n'en a pas précisé de nouveau (on aurait pu indiquer Copies/nouveau_nom)
- **ls nom_dossier** : affiche le contenu du dossier indiqué en argument

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rm cal_fact3.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
. .. cal_fact2.adb cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ mv cal_fact2.adb cal_fac
t_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
. .. cal_fact_nouv.adb cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ mkdir Copies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ mv cal_fact_nouv.adb Cop
ies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
. .. cal_factorielle.adb Copies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls Copies
. .. cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

- la commande **cp -R dossier_source dossier_destination** permet de copier de façon récursive un dossier dans un autre. Ici, on copie le dossier source « Copies » dans un nouveau dossier « Copies2 », et on vérifie si cela a été correctement réalisé.
- Si le nom du dossier destination existe, la copie se fait dans ce dossier existant. Ici la deuxième commande cp copie le dossier source « Copies » dans le dossier existant « Copies2 » en en gardant le même nom (mais on aurait pu en indiquer un : Copies2/nouveau_nom)

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_factorielle.adb  Copies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls Copies
.  ..  cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ cp -R Copies Copies2
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_factorielle.adb  Copies  Copies2
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls Copies2
.  ..  cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ cp -R Copies Copies2
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_factorielle.adb  Copies  Copies2
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls Copies2
.  ..  cal_fact_nouv.adb  Copies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

- La commande **rmdir** permet de détruire un dossier vide : ici la tentative de détruire le dossier « Copies2/Copies » échoue car ce dossier n'est pas vide
- On vide donc le dossier « Copies2/Copies » (destruction du seul fichier qu'il contient) et on exécute de nouveau la commande **rmdir**, et on vérifie que le dossier « Copies » a disparu du dossier « Copies2 »
- Pour détruire un dossier non vide, on peut utiliser la commande **rm -R** mais il faut être sûr de ce qu'on fait. Pour plus de sécurité, on peut utiliser l'option **-i** pour demander confirmation et on répond par « y » ou « n ».

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rm Copies2/Copies
rm: impossible de supprimer 'Copies2/Copies': est un dossier
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rmdir Copies2/Copies
rmdir: impossible de supprimer 'Copies2/Copies': Le dossier n'est pas
vide
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rm Copies2/Copies/cal_fa
ct_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rmdir Copies2/Copies
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls Copies2
.  ..  cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ rm -Ri Copies2
rm : descendre dans le répertoire 'Copies2' ? y
rm : supprimer 'Copies2/cal_fact_nouv.adb' du type fichier ? y
rm : supprimer 'Copies2' du type répertoire ? y
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

- La commande **ls -R** affiche le contenu de toute l'arborescence d'un répertoire
- La commande **wc** affiche le nombre de ligne, de mots et de caractères du fichier donné en argument : ici **cal_factorielle.adb** en contient respectivement 21, 86 et 675.
- Avec l'option **-l** **wc** n'affiche que le nombre de ligne (**-w** pour les mots et **-c** pour les caractères)

- La commande `grep` permet d'afficher les lignes qui contiennent un motif, ici le mot « begin », en indiquant le numéro de la ligne si l'option `-n` est précisée.

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -R
.:
.  ..  cal_factorielle.adb  Copies

./Copies:
.  ..  cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ wc cal_factorielle.adb
 21  86 675 cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ wc -l cal_factorielle.adb
21 cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ grep -n begin cal_factorielle.adb
8:begin
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

4- Caractères Joker

Le shell offre de nombreuses facilités, parmi lesquelles l'utilisation de caractères joker :

- `*` désigne n'importe quelle suite de caractères : dans le terminal dessous, `ls *` affiche tous les objets du répertoire courant et leur contenu lorsqu'ils sont des dossiers, comme si on avait exécuté `ls cal_fact2.adb cal_factorielle.adb factorielle.adb Copies`
- `ls *.adb` ne liste que les objets se terminant par `.adb`
- `ls cal*.adb` ne liste que les fichiers commençant par `cal` et se terminant par `.adb`
- `?` désigne un seul caractère quelconque : `ls cal_fact?.adb` ne liste que le fichier `cal_fact2.adb`, seul correspondant au motif indiqué
- On peut préciser un ensemble de caractères motifs en les mettant entre `[...]` :
`ls cal_fact[23]*` n'affiche que les fichiers qui contiennent les caractères 2 ou 3 après `cal_fact`
- Un message d'erreur est affiché lorsqu'aucun objet ne correspond au motif indiqué.

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls
.  ..  cal_fact2.adb  cal_factorielle.adb  Copies  factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls *
cal_fact2.adb  cal_factorielle.adb  factorielle.adb

Copies:
.  ..  cal_fact_nouv.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls *.adb
cal_fact2.adb  cal_factorielle.adb  factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls cal*.adb
cal_fact2.adb  cal_factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls cal_fact?.adb
cal_fact2.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls *.c
ls: impossible d'accéder à '*.c': Aucun fichier ou dossier de ce type
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

5- Droits d'accès

Sous Unix/Linux les objets possèdent 3 droits d'accès : lecture, écriture et exécution ; et chacun de ses 3 droits est décliné en 3 niveaux : pour le propriétaire, le groupe et les autres utilisateurs. Ces droits sont visibles lorsqu'on liste les objets avec l'option `-l`, sous la forme de 9 caractères `rw-rw-rw-` : les 3 premiers pour le propriétaire, les 3 suivants pour le groupe et les derniers pour les autres. Lorsqu'un caractère est présent (r pour read, w pour write et x pour exec) il indique que le droit correspondant est ouvert, sinon c'est un « - » qui est présent.

Dans le terminal suivant, on voit que :

- Les fichiers `.adb` sont accessibles en lecture et écriture pour le propriétaire, seulement en lecture pour le groupe, et ne sont pas accessibles aux autres. Ils ne sont pas exécutables, donc le caractère « x » est absent.
- Les dossiers « . » (courant), « .. » (parent), et Copies, 1^{er} champ commençant par d, sont accessibles en exécution (x) pour tous (cela veut dire que l'on peut aller dedans), sont lisibles (r) par le propriétaire et le groupe (on peut lister leur contenu), mais ne sont modifiables (w) que par le propriétaire (qui peut y ajouter des objets).

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -l
total 24
drwxr-xr-x 3 hamrouni gea 4096 sept.  4 14:30 .
drwxr-xr-x 3 hamrouni gea 4096 sept.  4 11:08 ..
-rw-r----- 1 hamrouni gea  675 sept.  4 13:28 cal_fact2.adb
-rw-r----- 1 hamrouni gea  675 sept.  4 11:12 cal_factorielle.adb
drwxr-xr-x 2 hamrouni gea 4096 sept.  4 12:04 Copies
-rw-r----- 1 hamrouni gea  675 sept.  4 13:29 factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

On peut modifier les droits d'accès en utilisant la commande `chmod nouveaux_droits fichier/dossier`. Les nouveaux droits peuvent être exprimés de deux manières :

- sous forme numérique (7 pour `rw-` (111 en binaire), 6 pour `rw-` (110 en binaire), ...)
- ou sous la forme d'ajout (+) ou de retrait (-).

Par exemple la commande `chmod g+w cal_fact2.adb` ajoute (+) au groupe (g) le droit d'écriture (w) sur le fichier en question, et la commande `chmod u-w` supprime le droit d'écriture au propriétaire (u : user, o : others).

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ chmod g+w cal_fact2.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ chmod u-w cal_fact2.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -l
total 24
drwxr-xr-x 3 hamrouni gea 4096 sept.  4 14:30 .
drwxr-xr-x 3 hamrouni gea 4096 sept.  4 11:08 ..
-r--rw---- 1 hamrouni gea  675 sept.  4 13:28 cal_fact2.adb
-rw-r----- 1 hamrouni gea  675 sept.  4 11:12 cal_factorielle.adb
drwxr-xr-x 2 hamrouni gea 4096 sept.  4 12:04 Copies
-rw-r----- 1 hamrouni gea  675 sept.  4 13:29 factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

6- Edition de fichiers

La commande « cat » (Concatenate) permet d'afficher dans le terminal le contenu des fichiers passés en arguments. Elle n'a d'intérêt que lorsque ce contenu est un texte simple, sans formatage. Les fichiers contenant du formatage (word, execl, pdf, ...) ont besoin des logiciels associés, et les fichiers binaires (exécutables par exemple) ne sont pas interprétables visuellement. Dans le terminal suivant, on ne voit que les dernières lignes du fichier `cal_factorielle.adb` affiché par `cat`.

```
end loop ;
New_Line ;
Put_Line("La factorielle de"& Integer'Image(entier) &" est"& Integer'Image(
orielle));
Put("Entrer la valeur dont vous voulez calculer la factorielle (>=0) : ") ;
Get(entier);
end loop ;
end cal_factorielle ;
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

La commande `cat` présente l'inconvénient d'afficher tout en un seul bloc quelle que soit la taille du terminal. Une commande différente « `more` » permet d'afficher par blocs en fonction de la taille du terminal, et peut s'avérer plus pratique dans certaines situations.

Mais pour pouvoir écrire du code et le modifier, on a besoin d'un éditeur texte. Il y en a des dizaines plus ou moins élaborés. Ils sont généralement graphiques et offrent des fonctionnalités accessibles avec un simple clic de la souris. Mais lorsqu'on est connecté à distance sans mode graphique, l'utilisation d'un éditeur graphique est impossible.

Il existe des éditeurs utilisables en mode commande, et peuvent s'avérer utiles lorsqu'on est connecté en mode minimal (terminal shell). « `vim` » en est un, et offre en plus de nombreux avantages qui rendent la manipulation du code très pratique. Il offre en plus une version graphique appelée « `gvim` », ce qui permet de réunir les avantages des deux modes.

La commande `vim cal_factorielle.adb` ouvre le fichier en argument dans l'éditeur « `vim` », dont on voit le début dans le terminal suivant. On peut remarquer toutes les utilités offertes par les bons éditeurs : coloration syntaxique, numérotation des lignes, indentation (décalage des lignes en fonction de la profondeur des structures de contrôle).

La dernière ligne du terminal sert d'interface avec l'utilisateur (affichage d'informations, saisie de commandes. Ici, `:1,20 s/entier/nombre/g` est une commande entrée par l'utilisateur (ne fait partie du fichier), et qui si elle est validée par « `return` » va remplacer toute occurrence du mot « `entier` » par « `nombre` » entre les lignes 1 et 20. Vous pouvez mesurer ainsi sa puissance, car avec un minimum de maîtrise les actions correspondantes deviennent plus rapides à exécuter qu'en mode graphique.

```
Bitwise xterm - hamrouni@c201-05.enseeiht.fr:22
1 with ada.text_io ; use ada.text_io ;
2 with ada.integer_text_io ; use ada.integer_text_io ;
3
4 procedure cal_factorielle IS
5
6   entier, factorielle: integer ;
7
8 begin
:1,20 s/entier/nombre/g
```

Vous apprendrez à le manipuler en TP, mais vous pourrez choisir d'autres solutions en fonction de votre intérêt et des ordinateurs sur lesquels vous travaillerez.

7- Compilation et exécution

Une fois le fichier source (code du programme) complété et enregistré, une étape de compilation est nécessaire (sauf pour les langages interprétés tel que python). Elle consiste à vérifier que le code respecte les règles syntaxique et lexicales du langage, puis à générer un programme binaire exécutable (traduction des instructions de haut niveau en instructions machines puis en code binaire).

Pour compiler un programme écrit en langage « ada », on utilise la commande « gnatmake », comme c'est montré dans le terminal suivant, où on voit que la commande, si elle réussit, génère différents fichiers dont le plus important est l'exécutable `cal_factorielle` (sans extension).

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ gnatmake cal_factorielle.adb
gcc -c cal_factorielle.adb
gnatbind -x cal_factorielle.ali
gnatlink cal_factorielle.ali
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ls -l
total 708
drwxr-xr-x 3 hamrouni gea 4096 sept. 4 15:15 .
drwxr-xr-x 3 hamrouni gea 4096 sept. 4 11:08 ..
-r--rw---- 1 hamrouni gea 675 sept. 4 13:28 cal_fact2.adb
-rwxr-xr-x 1 hamrouni gea 691464 sept. 4 15:15 cal_factorielle
-rw-r----- 1 hamrouni gea 675 sept. 4 11:12 cal_factorielle.adb
-rw-r--r-- 1 hamrouni gea 2485 sept. 4 15:15 cal_factorielle.ali
-rw-r--r-- 1 hamrouni gea 3256 sept. 4 15:15 cal_factorielle.o
drwxr-xr-x 2 hamrouni gea 4096 sept. 4 12:04 Copies
-rw-r----- 1 hamrouni gea 675 sept. 4 13:29 factorielle.adb
hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

On voit bien qu'il possède l'attribut « x », donc exécutable, et qu'il a un volume 10 fois supérieur à celui du fichier source. Les fichiers exécutables sont volumineux et doivent être supprimés lorsqu'ils ne sont plus utiles, d'autant plus qu'ils peuvent être régénérés à tout moment à partir du code source.

Pour exécuter ce code, on lance le fichier exécutable (comme une commande) en le faisant précéder par « ./ » pour dire qu'il se trouve dans le répertoire courant. Le programme demande à l'utilisateur d'entrer une valeur, en calcule et affiche la factorielle si elle est ≥ 0 , et s'arrête si elle est négative.

Le terminal suivant montre une exécution de ce programme :

- l'utilisateur entre la valeur 5, et voit le résultat s'afficher
- puis entre la valeur 0, et voit le résultat s'afficher
- et finit par entrer -1 pour indiquer au programme qu'il souhaite arrêter

```
hamrouni@gobelin:~/Env_Info_Demo/Programmes$ ./cal_factorielle
Entrer la valeur dont vous voulez calculer la factorielle ( $\geq 0$ ) : 5

La factorielle de 5 est 120
Entrer la valeur dont vous voulez calculer la factorielle ( $\geq 0$ ) : 0

La factorielle de 0 est 1
Entrer la valeur dont vous voulez calculer la factorielle ( $\geq 0$ ) : -1

hamrouni@gobelin:~/Env_Info_Demo/Programmes$
```

8- Processus

Un programme en cours d'exécution constitue une entité appelée « processus », et qui est composée du code, des ressources nécessaires allouées par le système (mémoire, descripteurs d'entrées/sorties, priorités, etc.). Sur un ordinateur, plusieurs dizaines, voire centaines, de processus s'exécutent en pseudo-parallélisme : l'utilisateur a l'impression que tous les processus sont actifs en même temps, mais en réalité le système divise le temps en petites périodes très fines, et alloue de façon tournante les ressources nécessaires à l'exécution du processus sélectionné. De façon simplifiée, un processus va s'exécuter durant X millisecondes, puis va être mis en attente durant N x X millisecondes avant d'avoir la main de nouveau.

Chaque processus possède un numéro attribué par le système (PID), et est rattaché à un père dont le numéro est noté PPID. Toute commande lancée depuis un terminal shell a pour père le processus associé au terminal.

La commande « **ps** » affiche la liste des processus en cours, sous différentes formes selon les options choisies. Dans le terminal suivant la commande « **ps l** » affiche la liste des processus lancés par l'utilisateur « hamrouni » dans sa session de travail sur la machine « phosphore ». On y voit :

- Un premier processus portant le PID 29080 et le nom « -tcsh » : c'est le processus associé à un premier terminal (pts/0)
- Un deuxième processus portant le PID 29083 et ayant pour père le processus 29080 (-tcsh) et portant le nom « bash » : c'est le bash lancé dans le terminal tcsh
- Le processus associé à un second terminal (PID=32318), et le bash lancé dedans (PID=32321)
- Le processus créé par la commande **ps l** (PID=32409), et que son père est le bash depuis lequel il a été lancé (PPID=32321)

```
Bitwise xterm - hamrouni@c202-07.enseeiht.fr:22
hamrouni@phosphore:~$ ps l
 F  UID    PID  PPID  PRI  NI   VSZ   RSS  WCHAN  STAT TTY      TIME COMMAND
 0  22052  29080  21347  20   0  45396  3828  sigsus  Ss   pts/0    0:00 -tcsh
 0  22052  29083  29080  20   0  39684  4656  poll_s  S+   pts/0    0:00 bash
 0  22052  32318  21347  20   0  45396  3852  sigsus  Ss   pts/1    0:00 -tcsh
 0  22052  32321  32318  20   0  39684  4452  wait    S    pts/1    0:00 bash
 0  22052  32409  32321  20   0  30236  1464  -       R+   pts/1    0:00 ps l
hamrouni@phosphore:~$
```

En lançant la commande **sleep 600** dans le premier terminal (commande qui va attendre 600 secondes avant de s'arrêter) et en exécutant de nouveau la commande **ps l** dans le second terminal, on voit apparaître un nouveau processus portant le PID 6478 lancé depuis le premier terminal (pts/0), et dont le père est le « bash » de pid 29083.

```
hamrouni@phosphore:~$ ps l
 F  UID    PID  PPID  PRI  NI   VSZ   RSS  WCHAN  STAT TTY      TIME COMMAND
 0  22052   6478  29083  20   0   8704   876  hrtime  S+   pts/0    0:00 sleep 600
 0  22052   6504  32321  20   0  30236  1464  -       R+   pts/1    0:00 ps l
 0  22052  29080  21347  20   0  45396  3828  sigsus  Ss   pts/0    0:00 -tcsh
 0  22052  29083  29080  20   0  40016  4832  wait    S    pts/0    0:00 bash
 0  22052  32318  21347  20   0  45396  3852  sigsus  Ss   pts/1    0:00 -tcsh
 0  22052  32321  32318  20   0  39684  4460  wait    S    pts/1    0:00 bash
hamrouni@phosphore:~$
```


La colonne « STATUS » indique l'état du processus :

- R (Running) : en cours d'exécution (cas de la commande ps l)
- S (Sleeping) : endormi (cas des processus bash et cal_factorielle)

Suspension

On peut suspendre un processus : il reste présent, mais ne peut pas reprendre la main.

Dans le terminal suivant (pts(1)), la commande **kill -TSTP 6478** envoie un signal de suspension (TSTP) au processus 6478 (sleep 600 lancé dans le premier terminal (pts/0)), et la commande ps l montre qu'il passe dans l'état T (sTopped, suspendu)

```
hamrouni@phosphore:~$ kill -TSTP 6478
hamrouni@phosphore:~$ ps l
F  UID    PID  PPID  PRI  NI   VSZ   RSS  WCHAN  STAT TTY      TIME COMMAND
0 22052  6478 29083  20   0   8704   876  signal  T    pts/0    0:00 sleep 600
0 22052  6915 32321  20   0  30236  1520  -       R+   pts/1    0:00 ps l
0 22052 29080 21347  20   0  45396  3828  sigsus  Ss   pts/0    0:00 -tcsh
0 22052 29083 29080  20   0  40016  4832  poll_s  S+   pts/0    0:00 bash
0 22052 32318 21347  20   0  45396  3852  sigsus  Ss   pts/1    0:00 -tcsh
0 22052 32321 32318  20   0  39684  4460  wait    S    pts/1    0:00 bash
hamrouni@phosphore:~$
```

En on voit dans le premier terminal (pts/0) un message indiquant que « sleep » a été arrêté, et que le prompt est libéré (on peut entrer de nouvelles commandes)

```
Bitwise xterm - hamrouni@c202-07.enseeiht.fr:22
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ sleep 600

[1]+  Arrêté                  sleep 600
hamrouni@phosphore:~/Env_Info_Demo/Programmes$
```

Reprise

On peut relancer un processus suspendu avec la commande **kill -CONT 6478** comme le montre le terminal suivant : le processus 6478 a été relancé et trouve dans l'état S (il n'a pas la main).

```
Bitwise xterm - hamrouni@c202-07.enseeiht.fr:22
hamrouni@phosphore:~$ kill -CONT 6478
hamrouni@phosphore:~$ ps l
F  UID    PID  PPID  PRI  NI   VSZ   RSS  WCHAN  STAT TTY      TIME COMMAND
0 22052  6478 29083  20   0   8704   876  restar  S    pts/0    0:00 sleep 600
0 22052  7444 32321  20   0  30236  1484  -       R+   pts/1    0:00 ps l
0 22052 29080 21347  20   0  45396  3828  sigsus  Ss   pts/0    0:00 -tcsh
0 22052 29083 29080  20   0  40016  4832  poll_s  S+   pts/0    0:00 bash
0 22052 32318 21347  20   0  45396  3852  sigsus  Ss   pts/1    0:00 -tcsh
0 22052 32321 32318  20   0  39684  4460  wait    S    pts/1    0:00 bash
hamrouni@phosphore:~$
```

Mais le processus 6478 ne reprend pas le contrôle de son terminal de lancement : il continue à s'exécuter en arrière-plan comme le montre la commande **jobs** qui affiche la liste des processus en arrière-plan. La commande **fg** ramène ce processus en avant plan.

```
[1]+  Arrêté                sleep 600
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ jobs
[1]+  En cours d'exécution  sleep 600 &
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ fg 1
sleep 600
```

Avant-plan et arrière-plan

La commande **fg** (foreground) permet de ramener un processus en avant-plan (prise de contrôle du terminal), en fournissant nom numéro d'ordre dans la liste « jobs » (et non son PID). Ici **fg 1** redonne le contrôle du terminal au processus sleep 600.

On peut dès le départ lancer un processus en arrière-plan en ajoutant l'option **&** en fin de commande. Ceci permet de laisser le terminal libre pour lancer d'autres commandes, mais ne permet pas au processus en question de lire des données au clavier (mais il peut afficher des messages) :

- **sleep 100 &** : présente un intérêt en lançant la commande sleep 100 en arrière-plan car ce processus n'effectue pas de lecture au clavier
- Mais Lancer cal_factorielle en arrière-plan n'a pas de sens, car dans ce cas les données lues au clavier sont fournies au shell qui aura repris le contrôle du terminal.

Arrêt d'un processus qui boucle

On peut se trouver dans la situation d'un programme qui boucle indéfiniment, et que l'on souhaite l'arrêter.

- Si le processus est en avant-plan dans un terminal, il suffit le plus souvent de taper en même temps les caractères Ctrl et C dans le terminal d'exécution. Dans le terminal suivant l'affichage du Ctrl C est remplacé par ^C et arrête le processus sleep 500

```
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ sleep 500
^C
hamrouni@phosphore:~/Env_Info_Demo/Programmes$
```

- Si le processus n'est pas rattaché à un terminal ou ne s'arrête pas avec Ctrl C, on peut l'arrêter de façon plus forte en lui envoyant le signal SIGKILL (9) : **kill -KILL num_proc** ou **kill -9 num_proc** comme le montrent les terminaux suivants :

```
hamrouni@phosphore:~$ ps 1
F  UID  PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0 22052 18781 29083 20   0    8704    836 hrtime  S+   pts/0      0:00 sleep 500
0 22052 18786 32321 20   0   30236   1512 -        R+   pts/1      0:00 ps 1
0 22052 29080 21347 20   0   45396   3828 sigsus  Ss   pts/0      0:00 -tcsh
0 22052 29083 29080 20   0   41044   6032 wait    S    pts/0      0:00 bash
0 22052 32318 21347 20   0   45396   3852 sigsus  Ss   pts/1      0:00 -tcsh
0 22052 32321 32318 20   0   39684   4460 wait    S    pts/1      0:00 bash
hamrouni@phosphore:~$ kill -KILL 18781
hamrouni@phosphore:~$
```

```
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ sleep 500
Processus arrêté
hamrouni@phosphore:~/Env_Info_Demo/Programmes$
```

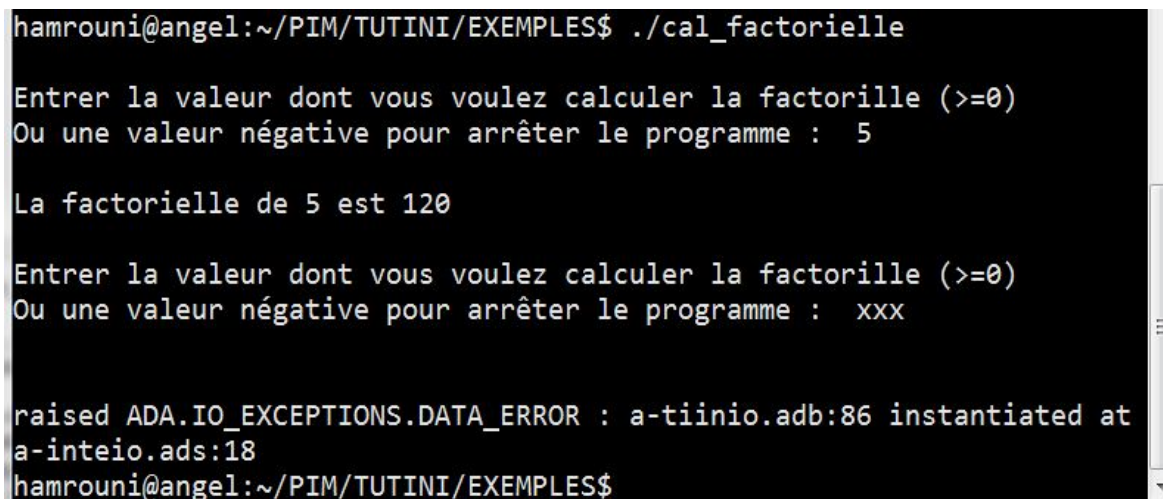
9- flots de données et redirections

Dans un processus sous Unix/Linux, les entrées/sorties sont gérées avec des descripteurs (numéros ≥ 0). Chaque processus en possède au minimum 3, ouverts automatiquement :

- Le descripteur 0 est associé à l'entrée standard « stdin » (clavier)
- Le descripteur 1 est associé à la sortie standard « stdout » (écran)
- Le descripteur 2 est associé à la sortie d'erreur « stderr » (écran)

Par exemple, le programme `cal_factorielle`, exécuté dans le terminal suivant :

- affiche sur la sortie standard le message demandant un nombre, en écrivant sur le descripteur 1 (Entrez la valeur dont ... »)
- lit sur l'entrée standard le nombre saisi par l'utilisateur en lisant sur le descripteur 0 (5)
- si l'utilisateur saisit une chaîne ne représentant pas un nombre, cela génère un message d'erreur qui est affiché sur la sortie d'erreur sur le descripteur 2 (« raised ADA.IO ... »)



```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle
Entrez la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : 5
La factorielle de 5 est 120
Entrez la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : xxx
raised ADA.IO_EXCEPTIONS.DATA_ERROR : a-tiinio.adb:86 instantiated at
a-inteio.ads:18
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Les échanges entre un processus et son environnement (entrées sorties : clavier, écran, fichiers, tubes de communication, sockets réseau, etc.) se font sous la forme d'un paquet d'octets, représentant des caractères ou sous forme binaire : On parle de **flots de données**.

Le système Linux offre un mécanisme simple permettant de rediriger les flots de données d'un processus sans en modifier le code. Et le shell permet de le mettre en œuvre de la façon suivante :

- commande `> fichier_resultats`
 - les messages standards sont enregistrés dans `fichier_resultats` au lieu d'être affichés à l'écran : on dit que la sortie standard (1) est redirigée vers `fichier_resultats` (vidé au préalable s'il existe déjà) : `ls > liste_fichiers` enregistre la liste des fichiers et dossiers du répertoire courant dans le fichier « `liste_fichiers` » au lieu de l'afficher à l'écran.
 - `>>` à la place de `>` : permet d'ajouter le flot de sortie au fichier (crée s'il n'existe pas)

- commande **2>** fichier_erreurs : les messages d'erreur sont enregistrés dans fichier_erreurs au lieu d'être affichés à l'écran : si on tape une commande qui n'existe pas XXX, un message d'erreur « La commande XXX n'a pas été trouvée ... » est affiché à l'écran. Si on veut ignorer ce message et ne pas le voir à l'écran, on peut taper : XXX 2> fic_erreurs, ce qui permet de rediriger le message d'erreur précédent dans le fichier « fic_erreurs ».
- commande **<** fichier_entrees : les entrées sont lues dans fichier_entrees et non au clavier

Nous allons illustrer cela de façon plus pratique et plus complète, en prenant comme exemple de base l'exécution du programme cal_factorielle. Le terminal suivant montre l'exécution de ce programme avec 3 valeurs entrées au clavier : 5, 10 et -1.

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : 5
La factorielle de 5 est 120
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : 10
La factorielle de 10 est 3628800
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : -1
Vous avez décidé d'arrêter
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Si l'utilisateur n'a pas envie de perdre son temps à entrer plusieurs valeurs au clavier, surtout si les valeurs sont nombreuses et si le temps de calcul entre 2 valeurs est long, on peut facilement enregistrer les différentes valeurs dans un fichier, et exécuter le programme en redirigeant l'entrée standard de manière à ce que le programme lise les nombres dans ce fichier et non au clavier. Le terminal suivant montre le contenu du fichier fic_entrees (préparé à l'avance) et la séquence d'exécution du programme cal_factorielle : les messages continuent à s'afficher à l'écran, mais l'utilisateur n'a plus besoin d'entrer ses valeurs au clavier, car elles sont lues dans fic_entrees :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ more fic_entrees
5
10
-1
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle <fic_entrees
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme :
La factorielle de 5 est 120
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme :
La factorielle de 10 est 3628800
Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : Vous avez décidé d'
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Il sera encore plus pratique de sauvegarder les résultats dans un fichier pour pouvoir les consulter ou les exploiter à tout moment sans avoir à exécuter de nouveau le programme. Le terminal suivant montre l'exécution de `cal_factorielle` avec redirection de l'entrée standard et de la sortie standard : aucune lecture n'est faite au clavier et aucun résultat n'est affiché à l'écran.

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle <fic_entrees
>fic_resultats
```

Et on peut consulter les résultats dans le fichier `fic_resultats` :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ more fic_resultats

Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme :
La factorielle de 5 est 120

Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme :
La factorielle de 10 est 3628800

Entrer la valeur dont vous voulez calculer la factorielle (>=0)
Ou une valeur négative pour arrêter le programme : Vous avez décidé
d'arrêter
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ grep ^La fic_resultats
La factorielle de 5 est 120
La factorielle de 10 est 3628800
```

Ci-dessous, la commande « `grep` » permet de sélectionner les lignes qui contiennent les valeurs résultats (mot « `La` » en début de ligne) : `grep ^La` permet de sélectionner les lignes qui contiennent le mot « `La` » en leur début.

Mais il peut arriver qu'une entrée soit fausse, ce qui peut engendrer des messages d'erreur. En oubliant la dernière valeur `-1` dans le fichier `fic_entrees`, le programme `cal_factorielle` lit les valeurs `5`, `10` et les caractères de fin de fichier, qui provoquent un message d'erreur car ils ne représentent pas un nombre. Et ce message d'erreur est affiché sur la sortie d'erreur (l'écran). Le terminal dessous illustre ce cas :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ vim fic_entrees
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ more fic_entrees
5
10
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle <fic_entrees
>fic_resultats

raised ADA.IO_EXCEPTIONS.END_ERROR : a-textio.adb:506
```

Et si on veut éviter cet affichage à l'écran, il suffit de rediriger la sortie d'erreur (2) vers un fichier, ici `fic_erreurs`, dans lequel le message d'erreur va être affiché :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle <fic_entrees
>fic_resultats 2>fic_erreurs
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ more fic_erreurs

raised ADA.IO_EXCEPTIONS.END_ERROR : a-textio.adb:506
```


Cet exemple illustre les facilités offertes par le système Unix/Linux : il n'est pas nécessaire de modifier un programme fonctionnant en mode interactif pour le faire tourner sur des données issues de fichiers tout en sauvegardant les résultats dans un fichier résultat.

La séparation de la sortie standard et la sortie d'erreur permettent de séparer les traitements associés à ces flots de données.

10- couplage par tubes

Dans de nombreuses situations, on a besoin d'utiliser le flot de données résultat d'une commande comme entrée d'une autre commande. Un exemple simple : compter le nombre de lignes d'un fichier qui contiennent un mot donné. On dispose d'une commande qui affiche les lignes contenant un mot donné (`grep`), et d'une autre qui compte le nombre de ligne d'un fichier (`wc -l`), et on pourrait penser à utiliser un fichier intermédiaire dans lequel on stocke le résultat de la première commande pour s'en servir comme entrée de la seconde. Dans le terminal suivant :

- on exécute la commande `grep LOOP cal_factorielle.adb` et on voit qu'il y a 4 lignes qui contiennent le mot `LOOP`
- on exécute la même commande en redirigeant le résultat vers le fichier « `fic_loop` »
- puis on exécute la commande `wc -l` sur le fichier « `fic_loop` », qui affiche le résultat 4

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ grep LOOP cal_factorielle.adb
LOOP
                                FOR i IN 2..entier LOOP
                                END LOOP ;
                                END LOOP ;
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ grep LOOP cal_factorielle.adb >
fic_loop
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ wc -l fic_loop
4 fic_loop
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Cette solution est correcte, mais pas très pratique car elle oblige l'utilisateur à manipuler un fichier pour stocker des données intermédiaires. Heureusement le système Unix/Linux offre une meilleure facilité : au lieu de stocker les données dans un fichier, on les fait passer dans un tube qui permet d'établir une communication entre la première et la seconde commande : les résultats de la première entrent dans le tube, et la sortie du tube alimente l'entrée de la seconde commande.

Le shell met en œuvre ce mécanisme de la façon suivante : `commande1 | commande2`

Le tube (ou pipe) est matérialisé par le caractère « `|` ».

Le terminal suivant montre l'utilisation d'un pipe pour relier les commandes `grep` et `wc -l`

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ grep LOOP cal_factorielle.adb | wc -l
4
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

On peut combiner pipes et redirections, comme le montre l'illustration dessous : seul les messages commençant par « `La` » sont affichés à l'écran :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle < fic_entrees 2>
fic_erreurs | grep ^La
La factorielle de 5 est 120
La factorielle de 10 est 3628800
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Ou : seuls les messages commençant par « La » sont enregistrés dans fic_resultats :

```
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ ./cal_factorielle < fic_entrees 2>
fic_erreurs | grep ^La > fic_resultats
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$ more fic_resultats
La factorielle de 5 est 120
La factorielle de 10 est 3628800
hamrouni@angel:~/PIM/TUTINI/EXEMPLES$
```

Un exemple complet : on souhaite afficher les factorielles des dix premiers entiers en utilisant le programme cal_factorielle, avec une seule ligne de commande et sans passer par des fichiers.

- La commande **echo 1 2 3 4 5 6 7 8 9 10** permet de générer un flot de données composé par les 10 nombres passés en arguments, flot affiché par défaut à l'écran mais qui peut être redirigé via un pipe pour servir comme entrée de la commande cal_factorielle
- Cette dernière génère un flot de données composé de messages dont on va sélectionner ceux qui commencent par « La » (La factorielle de 5 est 120), en appliquant dessus la commande grep ^La
- Sur chaque message obtenu après le grep, on peut sélectionner le 6^{ème} champ pour ne garder que la valeur de la factorielle. La commande **cut -f6 -d' '** peut faire ce travail : -f6 sélectionne le champ numéro 6, et -d' ' précise que c'est l'espace ' ' qui est utilisé comme séparateur des champs.

```
hamrouni@boole:~/PIM/TUTINI/EXEMPLES$ echo 1 2 3 4 5 6 7 8 9 10 11 12 -1 |
./factorielle | grep ^La | cut -f6 -d' '
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
hamrouni@boole:~/PIM/TUTINI/EXEMPLES$
```

11- Quelques commandes avancées

La commande find permet d'effectuer des recherches dans une arborescence à partir d'un répertoire racine, et d'effectuer certains traitements sur les objets trouvés.

find [rep] [-name ...] [-type ...] [-size ...] [-ctime ...] [-mtime ...] [-atime ...] [-newer fichier]

[-print] [-exec cmd {} \;]

- Recherche récursivement :
 - depuis le répertoire « rep »
 - les fichiers dont le nom correspond au motif du `-name`
 - de type correspondant au motif de `-type (d/f/l)`
 - de taille correspondant au motif de `-size`
 - dont la date de (création, modification ou accès) correspond au motif `(+ -n : n nombre de jours)`
 - Plus récent que le fichier argument de `-newer`
- Affiche le nom du fichier (`-print`)
- Exécute la commande qui suit `-exec` en substituant `{}` par le nom du fichier. `\;` délimite la fin du `exec`

Dans le terminal suivant :

- La 1^{ère} commande recherche dans l'arborescence du répertoire Programmes tous les fichiers dont le nom se termine par `.adb`
- Le 2^{ème} recherche les fichiers dont le nom se termine par `.adb` et qui sont plus récents que `cal_fact2.adb`
- La 3^{ème} recherche tous les fichiers (sans motif de nom) dont la taille est supérieure à 1000 blocs (1 bloc = 512 octets) : la taille de l'exécutable `cal_factorielle` est de plus de 690 KO.

```
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -name '*.adb' -print
./cal_factorielle.adb
./Copies/cal_fact_nouv.adb
./factorielle.adb
./cal_fact2.adb
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -name '*.adb' -newer cal_fact2.adb
./factorielle.adb
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -size +1000
./cal_factorielle
hamrouni@phosphore:~/Env_Info_Demo/Programmes$
```

Dans le terminal suivant :

- La 1^{ère} commande recherche dans l'arborescence du répertoire Programmes tous les dossiers (`-type d`)
- Le 2^{ème} recherche les objets dont la date de création est supérieure à 3 jours
- Le 3^{ème} recherche les objets dont la date de création est inférieure à 3 jours
- Le 3^{ème} recherche les objets dont la date d'accès est inférieure à 3 jours

```

hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -type d
.
./Copies
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -ctime +3
./cal_factorielle.adb
./Copies
./Copies/cal_fact_nouv.adb
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -ctime -3
.
./factorielle.adb
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -atime -3
.
./cal_factorielle
./Copies
./factorielle.adb
hamrouni@phosphore:~/Env_Info_Demo/Programmes$

```

Dans le terminal suivant :

- La 1^{ère} commande recherche dans l'arborescence du répertoire Programmes tous les fichiers (sans motif de nom) dont la taille est supérieure à 1000 blocs (1 bloc = 512 octets), et exécute dessus la commande ls -l (affichage long)
- La 2^{ème} commande fait le même travail, mais en passant par un pipe et la commande xargs : cette dernière découpe le flot de données fourni par la 1^{ère} commande (find) et fournit chaque élément comme argument à la commande suivante (ls -l)
- La 3^{ème} commande effectue la même recherche et exécute sur chaque fichier trouvé la commande mv pour le déplacer dans le répertoire ~/nosave/Poubelle. On voit bien que le fichier cal_factorielle n'existe plus dans le répertoire Programmes. On ne peut pas utiliser xargs à la place de -exec, car la commande mv prend 2 arguments et xargs ne saura pas les distinguer.

```

hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -type f -size +1000
-exec ls -l {} \;
-rwxr-xr-x 1 hamrouni gea 691464 sept.  8 14:27 ./cal_factorielle
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -type f -size +1000
|xargs ls -l
-rwxr-xr-x 1 hamrouni gea 691464 sept.  8 14:27 ./cal_factorielle
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -type f -size +1000
-exec mv {} ~/nosave/Poubelle/ \;
hamrouni@phosphore:~/Env_Info_Demo/Programmes$ find . -type f -size +1000
hamrouni@phosphore:~/Env_Info_Demo/Programmes$

```

12. Travail à distance

Vous pouvez travailler depuis chez-vous en vous connectant sur la plateforme : <https://vclass.inp-toulouse.fr/> qui donne accès à différentes ressources logicielles via le navigateur web (html) ou en installant un client. Vous pourrez ainsi accéder à une station virtuelle linux (même environnement que celui des machines N7, sauf quelques exceptions), et à vos données, et travailler comme si vous étiez dans une salle de TP de l'N7.

En cas de saturation ou d'indisponibilité de cette plateforme, vous pourrez installer sur votre ordinateur un client ssh (par exemple bitwiseSSH pour windows), ou x2go qui prend en charge différents environnements graphique, contrairement à ssh. L'utilisation du VPN est nécessaire dans ce cas. Voir le document spécifique à la connexion à distance sur moodle.