INSTITUT Mines-Télécom

TELECOM SudParis

CSC 3102 – Sujets de TP Introduction aux systèmes d'exploitation

Année 2020-2021

Coordinateurs : Élisabeth Brunet et Amina Guermouche

http://www-inf.telecom-sudparis.eu/COURS/CSC3102/Supports/

CSC 3102 – Introduction aux systèmes d'exploitation

Portail informatique

Devoir Hors Présentiel – Calculette en Script shell

Objectifs : *Cet exercice a pour but de vous familiariser à la manipulation des structures de base du script shell afin que leur utilisation ne soit pas un frein à la réalisation des séances de TP suivantes.*

L'exercice consiste à réaliser une calculette en ligne de commande. Pour cela, nous procédons en plusieurs étapes. Dans la première version de notre calculette, l'objectif est d'appliquer un même opérateur aux opérandes donnés en argument de votre script. Dans une seconde version, votre calculette évoluera afin de permettre l'évaluation d'une suite d'opérandes et d'opérateurs sans priorité.

Avant de commencer, nous vous rappelons que le shell ne manipule que des chaînes de caractères. Ainsi, pour lancer l'évaluation d'une opération arithmétique, vous devez utiliser la commande **`expr`**.

Remarque : *Testez régulièrement vos avancées en lançant vos scripts avec différentes configurations d'arguments. Vous pouvez même scripter ces différents appels au sein d'un script shell indépendant afin d'automatiser vos tests. Dans tous les cas, n'oubliez pas de les rendre exécutables grâce à la commande* ***`chmod u+x <mon_script.sh>`****.*

Remarque : *Tous les exercices sont obligatoires.*

Remarque : *La charge de travail personnel prévisionnelle pour ce DM est estimée à 3h. Le devoir doit être rendu via la section dédiée sur la page moodle du cours. Vous êtes fortement incités à le réaliser en autonomie afin de vous familiariser avec les scripts shell. Vous êtes néanmoins autorisés à le réaliser en binôme (dans ce cas, l'un doit rendre le devoir tandis que l'autre doit rendre un fichier indiquant uniquement le nom du binôme avec qui il a travaillé).*

Bon travail !

Exercice 1 : Calculette uni-opération (~ 6 points)

Dans cette première phase, il s'agit de réaliser un script shell qui permette de réaliser une addition, une soustration, une multiplication ou une division sur l'ensemble des opérandes donnés en ligne de commande. Voici un exemple de résultat d'exécution attendu (les cinq derniers appels doivent être considérés par votre script comme incorrects et génèrer des messages d'erreurs) :

```
$./calculette_uni_operation.sh add 1 2 3
1 + 2 + 3 = 6
$./calculette_uni_operation.sh supp 6 4
6 - 4 = 2
$./calculette_uni_operation.sh mult 2 4 3
2 * 4 * 3 = 24
$./calculette_uni_operation.sh div 66 11 3 3
66 / 11 / 3 / 3 = 0
$
```

```
$
$./calculette_uni_operation.sh
Opération manquante
$./calculette_uni_operation.sh add
Opérande manquant
$./calculette_uni_operation.sh ad 1 2 3
Opération inconnue
$./calculette_uni_operation.sh 1 2 3
Opération inconnue
$./calculette_uni_operation.sh div 6 2 0
6 / 2 / 0 : Division par 0 interdite
$
```

Question 1.a. Commencez par consulter la page de manuel de la commande **`expr`** et testez-la.

Question 1.b. Supposez que l'appel à votre script est correct et traitez le cas de la somme. Pour vous aider à débuter, voici le pseudo code du script shell attendu.

```
        #! /bin/sh

        # déclaration d'une variable res dans laquelle le résultat sera accumulé, initialisée
avec le deuxième argument de la ligne
        # si le premier argument est égal à "add" alors
        #  décalage de deux arguments puisqu'on a traité l'opérateur et le premier opérande
        #  pour chaque argument suivant
        #    res=res+argument courant
        #    affichage de l'opération effectuée
        #
        # affichage du résultat
```

Attention : ***`res`*** *ne doit pas être égal à la chaîne de caractère* ***`<res+argument courant>`****, mais le résultat de l'évaluation de cette commande.*

On vous rappelle donc que, comme vu en cours, vous pouvez affecter une variable à la sortie d'une commande avec la construction ***`bash`*** *suivante :*

```
var=$(cmd arg1 arg2...)
```

Par exemple, ***`x=$(expr 1 + 2)`*** *affectera la valeur* ***`3`*** *à la variable* ***`x`****.*

Question 1.c. Mettez en place un schéma alternatif afin de traiter les différents opérateurs. Attention à protéger correctement le caractère spécial '*' dans l'évaluation de la multiplication.

Question 1.d. Traitez le cas de la division par zéro.

Question 1.e. À présent, traitons les cas d'erreurs. Voici le pseudo code du script attendu :

```
        #! /bin/sh

        #si le nombre d'arguments est égal à 0
```

```
          #  affichage "Opération manquante"
          #  sortie du script
          #sinon
          #  si le nombre d'arguments est égal à 1
          #    affichage "Opérande manquant"
          #    sortie du script
          #  sinon
          #    res=deuxième argument de la ligne
          #    affichage
          #    si le premier argument est égal à
          #      "add" :
           #         décalage de deux arguments puisqu'on a traité l'opérateur et le premier
opérande
          #        pour chaque argument suivant
          #        res=res+argument courant
          #        affichage de l'opération effectuée
          #      "supp" :
          #        ...
          #      "mult" :
          #        ...
          #      "div" :
          #        si operande = 0
          #          affichage " / 0 : Division par 0 interdite"
          #        sinon
          #          ...
          #     autre cas :
          #        affichage "Operation non traitée"
          #        sortie du script
          #    affichage du resultat
```

Exercice 2 : Calculette multi-opération sans priorité (~ 10 points).

À présent, nous vous proposons de réaliser une calculette qui permet d'appliquer les opérations arithmétiques dans l'ordre où elles sont écrites, i.e. sans tenir compte des priorités sur les opérateurs. Voici un exemple de résultat d'exécution attendu :

```
$./calculette_sans_priorite.sh 1 + 2 + 3
1 + 2 + 3 = 6
$./calculette_sans_priorité.sh 6 - 4
6 - 4 = 2
$./calculette_sans_priorité.sh 2 x 4 + 3
2 * 4 + 3 = 11
$./calculette_sans_priorite.sh 3 + 2 x 4
3 + 2 * 4 = 20
$./calculette_sans_priorite.sh 3 + 2 x 4 / 2
3 + 2 * 4 / 2 = 10
$
$
$./calculette_sans_priorite.sh
```

```
Opérande manquant
$./calculette_sans_priorite.sh 1 +2
1 +2 -> Opération non traitée
$./calculette_sans_priorite.sh 1 / 0
1 /0 -> Division par 0 interdite
$
```

Remarque :

- *Afin de pas à se soucier de l'interprétation du symbole '**' en tant qu'opérateur de multiplication de notre script shell, utilisez le caractère **"x"** à la place.*
- *Attention à bien séparer vos arguments par des espaces.*

Question 2.a. Dans un nouveau script shell, abordez le problème sans vous soucier des cas d'erreurs. Pour vous aider, voici le pseudo code du script shell attendu :

```
#! /bin/sh
#
#res=premier argument
#affichage
#décalage
#
#tant qu'il reste des arguments
#  si l'argument courant est égal à
#  + :
#   res=res+ argument qui suit
#   affichage
#  - :
#   ...
#  x :
#   ...
#  / :
#   si argument qui suit = 0, affichage "Divivion par 0 interdite"
#
#  décalage
#affichage du résultat
```

Question 2.b. Traitez les cas d'erreurs.

Exercice 3 : Calculette multi-opération avec priorité (~ 4 points)

En vous basant sur le script précédent, faites en sorte de prendre en compte les priorités qu'imposent les opérateurs arithmétiques, et ce, sans parenthèse.

La multiplication et la division sont prioritaires sur l'addition et la soustraction; que ce soit dans une suite d'additions et de soustractions ou dans une suite de multiplications et de divisions, on effectue les calculs dans l'ordre d'écriture (c'est-à-dire de manière similaire à la question précédente). Ainsi, le challenge réside dans la différenciation des opérations selon leur priorité.

Pour vous aider, à chaque fois que vous rencontrez un opérateur non prioritaire, il faut que vous détectiez si l'opération suivante doit être évaluée avant. Plusieurs opérations prioritaires pouvant se suivre, cette vérification + évaluation doit être

faite au sein d'une boucle qui prend fin lorsque l'opérateur rencontré n'est ni un opérateur de multiplication ni un opérateur de division.

Trucs et astuces

Cette petite liste de trucs et astuces devrait vous aider à utiliser un peu plus efficacement votre terminal et le langage **bash**. Lisez attentivement cette page, vous gagnerez beaucoup de temps par la suite !

Exercice 1 : Questions fréquentes

Question 1.a. bash: [: =: unary operator expected

Question 1.b. Comment me connecter aux salles de TP depuis chez moi ?

Question 1.c. Je suis sous windows, comment je peux faire les TPs ?

Question 1.d. Je suis sous mac, comment je peux faire les TPs ?

Question 1.e. Quelle distribution de Linux je peux installer sur ma machine ?

Question 1.f. Comment utiliser emacs ?

Exercice 2 : Complétion des noms de fichiers

Pour utiliser la complétion automatique fournie par le shell, tapez les premières lettres de la commande ou du fichier et appuyez sur la touche « **`tabulation`** ». Si le préfixe donné n'est pas ambigu, la complétion est faite automatiquement. Sinon, si vous appuyez une deuxième fois sur la touche tabulation, le shell vous propose une liste des complétions disponibles.

Exercice 3 : Rappel des commandes précédentes

Pour rappeler des commandes précédentes dans votre fenêtre de terminal, vous pouvez presser la touche « **`flèche vers le haut`** » de votre clavier, une fois pour rappeler la commande précédente ou plusieurs fois pour rappeler une commande plus ancienne.

Lorsqu'une commande échoue, habituez-vous à utiliser l'historique pour la rappeler et la modifier au lieu de la retaper.

Vous pouvez aussi rechercher facilement une commande dans l'historique. Pour cela, il suffit de saisir la combinaison de touches « **`control + r`** ». Saisissez ensuite un des mots utilisé dans une ancienne commande et la commande complète va apparaître.

Exercice 4 : Ma commande est bloquée

Lorsque votre commande est bloquée (typiquement, lors d'une boucle infinie), vous pouvez interrompre la commande en utilisant la combinaison de touches « **`control + c`** ».

Exercice 5 : Mon terminal est bloqué

Votre terminal est bloqué car un processus tourne en premier plan. Vous pouvez soit interrompre le processus en premier plan en saisissant « **`control + c`** », comme indiqué dans l'astuce précédente, soit passer le processus en premier plan en arrière plan. Pour cela, il faut d'abord saisir la combinaison de touches « **`control + z`** » dans le terminal. Cette combinaison de touches suspend le processus en premier plan. Ensuite, vous pouvez redémarrer le processus en arrière-plan avec la commande **`bg`** (comme *background*).

Exercice 6 : Copier-coller avec la souris

Pour copier-coller une chaîne de caractères, sélectionnez-la avec le bouton gauche de la souris (ce qui copie), et collez-la avec le bouton du milieu (les deux boutons simultanément si la souris a seulement deux boutons). Le collage a lieu à la position courante du curseur.

Exercice 7 : Autres raccourcis `bash` bien utiles

De nombreux raccourcis de **emacs** fonctionnent aussi dans le terminal. En particulier :

- « **control + k** » : coupe du curseur jusqu'à la fin de la ligne.
- « **control + w** » : coupe du curseur jusqu'au début du mot précédent.
- « **control + y** » : colle le dernier texte coupé.

Exercice 8 : À propose de `source script.sh` et `. script.sh`

Pour les curieux, vous pouvez inclure un script dans un autre à l'aide de la commande **source** qui peut aussi s'écrire « **.** ». À ce moment, **bash** exécute les commandes se trouvant dans **script.sh**, comme si elles avaient été placées à la place de la ligne **source script.sh**. Ce mécanisme est très différent du lancement d'une commande en omettant le **source**. En effet, sans le **source**, **bash** lance un nouveau processus pour exécuter les commandes, alors qu'avec **source**, **bash** exécute le script passé en argument dans le processus courant. La construction **source** n'est pas utilisée dans le cours. En revanche, si vous ouvrez votre fichier de configuration par défaut (**.bashrc**), il est probable que vous croisiez une ligne contenant **. /etc/bashrc**. Cette ligne permet d'exécuter l'ensemble de commandes par défaut du système se trouvant dans le fichier **/etc/bashrc**.

TP1 – Bash

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs : *Le but de cette série d'exercices est de mettre en œuvre vos premiers scripts* ***`bash`****. Avant de pouvoir les exécuter, il faut au préalable les rendre exécutables avec la commande* ***`chmod a+x ./mon-script.sh`****. Cette notion vous sera expliquée dans les prochains cours.*

Exercice 1 : Guide de survie (~10mn)

Le but de cet exercice est de comprendre comment trouver de l'aide sur les commandes étudiées pendant le module.

Question 1.a. Ouvrez un terminal.

Question 1.b. Consultez la documentation de la commande **`hostname`**. À quoi sert-elle ?

Question 1.c. Exécutez la commande **`hostname`**. Vérifiez qu'elle affiche bien le nom de votre machine. Normalement, celui-ci est inscrit sur chaque tour ou écran en salle machine, suit le modèle <nom_salle>–<numero_machine>. Par exemple, **`b02–06`**, si vous êtes connecté sur la 6e machine de la salle b02.

Question 1.d. Consultez la documentation des commandes **`users`** et **`who`**.

Question 1.e. Utilisez ces commandes pour afficher votre identifiant de connexion.

Exercice 2 : Mon premier script shell (~15mn)

De façon à mettre en œuvre votre premier script shell, il faut démarrer un éditeur de texte. Nous vous conseillons fortement d'utiliser **`emacs`**. Si vous programmez déjà et avez l'habitude d'un autre outil comme **`vi`**, vous pouvez l'utiliser. Il vous est demandé de ne pas utiliser **`gedit`** ou **`nano`** qui sont particulièrement inadéquat dans ce cours. N'utilisez aussi surtout pas de traitement de texte de type **`Word`** ou **`Libreoffice`**, car ces derniers sauvegardent les fichiers dans des formats non reconnus par **`bash`**.

Question 2.a. Lancez un éditeur de texte.

Question 2.b. Dans l'éditeur de texte, programmez un script shell qui permet d'afficher la chaîne « **`Bonjour à tous !`** ». Vous nommerez ce script **`bonjour.sh`**.

Remarque : *Sur certaines distributions GNU/Linux, vous pouvez omettre le «* ***`#! /bin/bash`*** *» en début du programme. Nous vous déconseillons fortement d'utiliser cette possibilité car vos scripts deviendraient alors non portables sur d'autres systèmes UNIX ou d'autres distributions GNU/Linux.*

Question 2.c. Rendez le script exécutable en lançant, dans le terminal, la commande donnée en cours (**`chmod u+x bonjour.sh`**).

Question 2.d. Exécutez votre script.

Si votre terminal souhaite le bonjour à tous, bravo ! Vous venez de programmer et d'exécuter votre premier script shell !

Exercice 3 : Manipulation des arguments (~45mn)

Le comportement d'un script dépend souvent de données extérieures. Ces données peuvent provenir de plusieurs sources : d'un passage d'arguments du script, d'une interaction avec l'utilisateur lui demandant une saisie manuelle, d'un fichiers,

etc... La suite de l'exercice vise à apprendre comment manipuler les arguments.

Question 3.a. Modifiez le script de l'exercice précédent de manière à ce qu'il complète l'affichage produit avec un argument passé sur la ligne de commande. L'affichage devra être, par exemple :

```
$ ./bonjour.sh Yoda
Bonjour Yoda !
```

Question 3.b. Modifiez votre script afin qu'il prenne en compte non plus un argument, mais tous les arguments donnés sur la ligne de commande. Par exemple :

```
$ ./bonjour.sh Daenerys Arya Cersei
Bonjour Daenerys !
Bonjour Arya !
Bonjour Cersei !
```

ou

```
$ ./bonjour.sh "Daenerys Targaryen" "Arya Stark" "Cersei Lannister"
Bonjour Daenerys Targaryen !
Bonjour Arya Stark !
Bonjour Cersei Lannister !
```

Voici le squelette de l'algorithme attendu :

```
#! /bin/sh
#
# pour chaque argument de la liste des arguments de la ligne de commande du script
#   afficher la chaîne Bonjour suivie de l'argument courant suivie de !
```

Question 3.c. Modifiez votre script afin qu'il affiche la chaîne « **`Bonjour Marcheur Blanc !`** » si aucun argument n'a été donné. Voici le squelette de l'algorithme attendu :

```
#! /bin/sh
#
# si le script n'a pas d'argument alors
#   afficher la chaîne Bonjour Marcheur Blanc !
# sinon
#   pour chaque argument de la liste des arguments de la ligne de commande du script
#     afficher la chaîne Bonjour suivie de l'argument courant suivie de !
```

Question 3.d. On souhaite maintenant rendre votre script mal élevé. Votre script ne saluera donc plus qu'une personne sur deux. Par exemple :

```
$ ./bonjour.sh Daenerys Mélisandre Margaery Cersei Arya
Bonjour Daenerys !
Bonjour Margaery !
Bonjour Arya !
```

Voici le squelette de l'algorithme attendu :

```
#! /bin/sh
#
# si le script n'a pas d'argument alors
#   afficher la chaîne Bonjour Marcheur Blanc !
# sinon
#   tant que le nombre d'arguments est strictement supérieur à 0
#     afficher la chaîne Bonjour suivie de l'argument courant suivie de !
#     décaler les arguments
#     décaler les arguments
#
```

Exercice 4 : Commande `echo` et les caractères spéciaux du shell (~10mn – hors présentiel)

Le but de cet exercice est de mettre en évidence l'interprétation des caractères spéciaux du shell avec à la commande **`echo`**, et de montrer comment désactiver leur interprétation.

Remarque : *Au besoin, vous pouvez interrompre une commande en faisant* ***`control+c`****.*

Question 4.a. Petit échauffement : affichage de chaînes sans caractère spécial.

1. Affichez une simple chaîne de caractères, par exemple **`Bonjour`**.
2. La commande **`echo`** permet d'afficher plusieurs chaînes successivement. Affichez « **`Bonjour tout le monde`** ».
3. Vous pouvez également afficher des caractères non alphabétiques (tant qu'ils ne font pas partie des caractères spéciaux du shell), comme les caractères de ponctuation, les chiffres, etc. Affichez la chaîne : « **`2 fois Bonjour tout le monde !`** ».

Question 4.b. Désactivation de l'interprétation d'un caractère spécial.

1. En utilisant le caractère de désactivation « **`\`** », affichez la chaîne de caractères : « **`Le # est populaire`** ».
2. De même, affichez la chaîne de caractères « **`Le # est plus populaire que le \`** ».
3. Toujours en utilisant le caractère de désactivation « **`\`** », affichez la chaîne de caractères « **`Le # est plus populaire que le \ mais bien moins que l'*`** ».

Question 4.c. Désactivation de l'interprétation de plusieurs caractères spéciaux.

Remarque : *Lorsque plusieurs caractères spéciaux sont présents, les quotes (« **`'`** » et « **`"`** ») permettent de désactiver certains caractères spéciaux sur un ensemble de caractères. Pour rappel, les simples quotes (**`'...'`**), les désactivent tous, tandis que les doubles quotes (**`"..."`**) ne laissent actifs que les caractères « **`$`** », « **`\`** » et « **`` ` ``** » (accent grave).*

1. En utilisant les simples puis les doubles *quotes*, affichez la chaîne de caractères : « **`Le # est moins populaire que le *`** ».
2. En utilisant les simples puis les doubles *quotes*, affichez la chaîne de caractères : « **`Le # est plus populaire que le \`** ».

TP2 – Le système de fichiers

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs : *Ce TP est axé sur l'utilisation des commandes de manipulation des fichiers.*

Exercice 1 : Organisation de votre compte (~15mn)

Vous êtes libre d'organiser votre compte comme bon vous semble. Cependant, même si pour l'instant vous ne possédez que peu de fichiers, nous vous encourageons à organiser dès à présent vos fichiers en répertoire. Ainsi, nous vous suggérons de créer un répertoire par module de votre cursus, lui-même organisé par séance de TP.

Question 1.a. Positionnez-vous dans votre répertoire de connexion.

Question 1.b. Créez un répertoire nommé **`CSC3102`**.

Question 1.c. Déplacez-vous dans ce répertoire.

Question 1.d. Si vous n'avez pas utilisé l'interface graphique de votre système d'exploitation afin de créer des répertoires spécifiques, les fichiers correspondant à la première séance de TP ainsi qu'au devoir hors présentiel se trouvent à la racine de votre compte.

1. Créez un répertoire **`TP1_PremieresCommandes`**.
2. Déplacez-y les fichiers correspondant.
3. Créez un répertoire **`DM`**.
4. Déplacez-y les fichiers correspondant.

Question 1.e. Pour la séance de TP d'aujourd'hui, créez dans le répertoire **`CSC3102`** un répertoire nommé **`TP2_FS`**.

Exercice 2 : Se positionner dans le système de fichiers (~30mn)

Objectifs : *Le but de cet exercice est de vous faire utiliser les commandes de base de manipulation de fichiers :* ***`cd`****,* ***`cp`****,* ***`mv`****,* ***`mkdir`*** *et* ***`rm`****.*

Question 2.a. Positionnez-vous dans votre répertoire de connexion.

Question 2.b. Affichez le contenu du répertoire courant.

Question 2.c. Déplacez-vous dans le dernier répertoire créé, c.-à-d. celui correspondant au répertoire du TP2.

Question 2.d. Affichez le chemin du répertoire courant.

Question 2.e. Affichez le contenu du répertoire parent du répertoire courant.

Question 2.f. À ce stade, vous devez vous trouvez dans le répertoire TP2_FS. Vérifiez que c'est bien le cas et déplacez-vous y le cas échéant.

Question 2.g. Sans vous déplacer, afficher le contenu de votre répertoire créé pour le TP1 en utilisant un chemin absolu.

Question 2.h. Sans vous déplacer, afficher le contenu de votre répertoire de connexion en utilisant un chemin relatif.

Question 2.i. Créez un répertoire nommé **`test`** dans le répertoire courant et déplacez-vous y.

Question 2.j. Copiez le fichier **`/etc/passwd`** dans le répertoire courant.

Question 2.k. Dupliquez le fichier **`passwd`** dans le répertoire courant avec comme nouveau nom de fichier **`passwd2`**.

Question 2.l. Regardez les numéros d'inodes de ces deux fichiers avec l'option **`-i`** de la commande **`ls`**. Expliquez pourquoi ces numéros sont différents.

Question 2.m. Renommez le dernier fichier **`passwd2`** en **`dup`**.

Question 2.n. Regardez le numéros d'inode du fichier **`dup`** et comparez-le à celui qu'avait le fichier **`passwd2`**. Expliquez pourquoi ces numéros d'inodes sont identiques.

Question 2.o. Supprimez le fichier **`passwd`** (celui de votre compte bien sûr, pas celui qui se trouve en **`/etc`**).

Question 2.p. Supprimez le répertoire **`test`**.

Exercice 3 : Droits d'accès (~25mn)

Dans cet exercice, pensez à vérifier à chaque fois que nécessaire si les droits d'accès sont correctement positionnés à l'aide de la commande **`ls -l`**.

Question 3.a. Créez le répertoire tmp sous votre répertoire **`TP2_FS`** et positionnez les droits d'accès à « **`rwx r-x ---`** ». À quoi correspondent ces droits ?

Question 3.b. Copiez le fichier **`/etc/hosts`** sous **`tmp`** avec comme nouveau nom **`liste_hosts`**. Positionnez les droits d'accès à **`rw- r-- ---`** et lisez son contenu .

Question 3.c. Retirez pour le propriétaire le droit en lecture de **`liste_hosts`** et essayez de relire son contenu.

Question 3.d. Retirez pour le propriétaire le droit en écriture de **`tmp`** et essayez de détruire **`liste_hosts`**.

Question 3.e. Retirez pour le propriétaire le droit en lecture de **`tmp`** et essayez de lister son contenu.

Question 3.f. Retirez pour le propriétaire le droit en exécution (**`x`**) de **`tmp`** et essayez de vous positionner sur ce répertoire.

On a aussi perdu le droit de traverser le répertoire : impossible de se positionner dedans.

Question 3.g. Positionnez pour le propriétaire les droits d'accès **`rwx`** du répertoire **`tmp`**.

Question 3.h. Effacez tout le contenu du répertoire **`tmp`** et le répertoire lui-même.

Exercice 4 : Liens directs et liens symboliques (~20mn)

Question 4.a. Placez-vous dans le répertoire correspondant à la séance de TP2.

Question 4.b. Copiez dans ce répertoire le fichier **`/etc/passwd`** et nommez-le **`my_passwd`**.

Question 4.c. Affichez le contenu du fichier **`my_passwd`**. Pour cela, utilisez la commande **`cat file`** qui permet d'afficher le contenu du fichier **`file`** sur le terminal.

Question 4.d. Créez un répertoire nommé **`Liens`** dans le répertoire courant.

Question 4.e. Déplacez-vous dans le répertoire **`Liens`**.

Question 4.f. Sans vous déplacer, créez dans le répertoire **`Liens`** le lien direct **`lien.txt`** sur le fichier de votre répertoire courant **`my_passwd`**.

Question 4.g. Utilisez la commande **`ls`** pour afficher le numéro d'inode des fichiers **`my_passwd`** et **`lien.txt`**. Qu'observez-vous ?

Question 4.h. Avec la commande **`cat`**, affichez le contenu du fichier **`lien.txt`**. Cela doit afficher le même contenu qu'à la question c.

Question 4.i. Toujours sans vous déplacer, créez dans le répertoire **Liens** le lien symbolique **lien_symb.txt** sur le fichier **my_passwd**. Utilisez un chemin relatif pour référencer le fichier **my_passwd**.

Question 4.j. Utilisez la commande **ls** pour afficher le numéro d'inode des fichiers **my_passwd** et **lien_symb.txt**. Qu'observez-vous ?.

Question 4.k. Affichez le contenu du fichier **lien_symb.txt**.

Question 4.l. Déplacez le fichier **my_passwd** dans le répertoire **Liens**.

Question 4.m. Affichez le contenu des fichiers **lien.txt** et **lien_symb.txt**. Expliquez.

Question 4.n. Le cas échéant, réparez les liens brisés.

Question 4.o. Déplacez-vous à la racine de votre compte et affichez le contenu de **lien_symb.txt**.

Question 4.p. Sans vous déplacer, déplacez le fichier **my_passwd** dans son répertoire parent.

Question 4.q. Affichez de nouveau le contenu de **lien_symb.txt**.

Exercice 5 : La commande ssh (~30mn – hors présentiel)

Cet exercice a pour but de vous présenter la commande **ssh**. Cette commande vous permet de lancer un shell sur une machine distante de façon sécurisée. Elle vous sera utile pour travailler sur les machines de l'école lorsque vous n'y êtes pas physiquement. Le but de cet exercice n'est pas de vous faire un cours complet sur **ssh**, pour cela, nous invitons à consulter le cours se trouvant ici.

Les salles machines sont toutes connectées derrière une passerelle nommée **ssh.imtbs-tsp.eu**. Cette passerelle laisse passer toutes les connexions sortantes mais bloque les connexions entrantes. Dans ce TP, nous allons faire « comme si » nous étions à l'extérieur du réseau de l'école : nous allons nous connecter à la passerelle avant de nous connecter à une des machines.

Question 5.a. Ouvrez un terminal et lancez la commande **hostname**. Quel est le nom de votre machine ?

Question 5.b. Avant d'aller plus loin, lancez la fabuleuse commande **xeyes**. Que fait cette commande ?

Question 5.c. Fermez le processus **xeyes** puis utilisez la commande **ssh** pour vous connecter à la passerelle. Pour vous guider, vous devriez écrire **ssh mon-login@ssh.imtbs-tsp.eu** pour vous connecter (pensez à utiliser la commande **man** pour savoir comment utiliser la commande **ssh** de façon plus avancée). Qu'affiche maintenant la commande **hostname**.

Remarque : Cette remarque est importante, conservez ces informations quelques part, car vous en aurez très probablement besoin plus tard dans vos études ou votre vie professionnelle. *Comme vous vous connectez pour la première fois à la passerelle,* ***ssh*** *vous pose une question.* ***Avant de répondre « yes », lisez attentivement ce que vous raconte ssh****. De façon générale, si l'authenticité d'une machine ne peut pas être établie, cela signifie que :*

- *la machine sur laquelle vous vous connectez n'est pas celle que vous croyez. Le service qui trouve une machine à partir de son nom a très probablement été piraté. Si vous avez un doute, fuyez pauvres fous ! Contactez d'urgence l'administrateur de la machine et surtout, ne répondez pas* ***yes*** *aveuglément.*
- *la machine sur laquelle vous vous connectez vient d'être mise à jour. Dans ce cas, son identité a changé. Après vous être assuré(e) que c'était le cas, vous pouvez détruire l'entrée correspondante dans le fichier* ***~/.ssh/known_hosts*** *en lançant* ***ssh-keygen -R hostname****, où* ***hostname*** *est le nom de la machine sur laquelle vous vous connectez. Vous pourrez ensuite vous reconnecter comme si c'était une nouvelle machine;*
- *vous vous connectez pour la première fois à la machine. Dans ce cas, vous pouvez répondre* ***yes*** *sans danger;*

Question 5.d. Lancez la commande **xeyes**. Que se passe-t-il ?

Question 5.e. Techniquement, votre shell s'exécute sur l'une des passerelles du campus alors que votre écran est connecté à votre machine. Le shell de la passerelle est incapable de trouver le serveur graphique de votre machine, ce qui explique que la commande **`xeyes`** ne puisse pas se connecter.

Déconnectez-vous de la passerelle avec la commande **`exit`**, puis reconnectez-vous à la passerelle en utilisant l'option **`-Y`** de **`ssh`**. Cette option demande explicitement à **`ssh`** d'indiquer au **`shell`** de la passerelle où se trouve le serveur graphique.

Que se passe-t-il lorsque vous lancez **`xeyes`** ?

Question 5.f. Demandez à votre voisin le nom de sa machine. À partir du **`shell`** s'exécutant sur la passerelle, connectez-vous (avec **`ssh -Y`**) sur la machine de votre voisin et lancez un **`xeyes`** à distance. Où apparaît **`xeyes`** ?

Question 5.g. Déconnectez-vous de la machine de votre voisin et de la passerelle en saisissant **`exit`** deux fois. Vous devriez donc maintenant travailler sur votre machine. Vérifiez que c'est le cas en lançant la commande **`hostname`**. Si vous voyez que vous n'êtes pas en train de travailler sur votre machine, saisissez **`exit`** autant de fois que nécessaire.

Question 5.h. Nous allons maintenant apprendre à copier des fichiers d'une machine à une autre en utilisant le protocole **`ssh`**. Avant de commencer, de façon à vous remettre en jambe, copiez le fichier **`/etc/passwd`** de votre machine dans votre répertoire de connexion (répertoire « **`~`** ») en utilisant la commande **`cp`**.

Question 5.i. Pour copier un fichier d'une machine à une autre, il faut remplacer la commande **`cp`** par la commande **`scp`**. Cette commande fonctionne à peu près comme la commande **`cp`** mais permet de spécifier une machine cible ou une machine destination en préfixant un des chemins par « **`mon-login@nom-machine:`** » (par exemple, avec le chemin **`thomas_g@stromboli.telecom-sudparis.eu:~/movies/chat-noir-chat-blanc.avi`**).

Copiez le fichier **`/etc/passwd`** de la machine **`ssh.imtbs-tsp.eu`** dans votre répertoire de connexion.

Remarque : *Ce qu'il faut retenir : lorsque vous travaillez à distance, vous devez d'abord vous connecter à une passerelle de l'école avec* ***`ssh -Y ssh.imtbs-tsp.eu`****. Si par hasard* ***`ssh`*** *vous indique que la machine ne peut pas être authentifiée alors que vous vous y étiez déjà connecté, méfiez-vous, l'identité de la machine a peut-être été usurpé. Ensuite,* ***il vous est demandé de ne jamais travailler directement sur la passerelle*** *de façon à ce qu'elle ne soit pas surchargée (imaginez 1000 étudiants en train de travailler sur la même machine). Connectez-vous donc à une autre machine, en sachant que les machines s'appellent* ***`salle-num`****, où* ***`salle`*** *est un numéro de salle et* ***`num`*** *un numéro de machine (par exemple* ***`b02-08`****).*

TP3 – Les flux

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs : *Manipuler les flux.*

Exercice 1 : Un simple compteur (~40mn)

Le but de cet exercice est d'écrire un script nommé **`cpt.sh`** qui incrémente chaque seconde le nombre contenu dans un fichier.

Le script que vous mettez en œuvre dans cet exercice utilise un idiome (c.-à-d. un patron de programmation **`bash`**) que vous allez souvent retrouver dans les exercices qui suivent. Ce canevas est le suivant :

```
#! /bin/bash

# traite le cas dans lequel les arguments du script sont erronés
if <les arguments sont incorrects>; then
  echo "Un message d'erreur adequat" >&2 # &2 est la sortie d'erreur
  exit 1                                 # un code de retour faux
fi

# ici, vous mettez le corps du script

# le script s'est exécuté correctement, le script renvoie un code de retour vrai
exit 0
```

Dans la question **a**, on vous demande de traiter le cas dans lequel les arguments sont erronés. Dans les questions **b** et **c**, on vous demande de remplir le corps du script : initialiser le fichier passé en argument en y écrivant un 0, puis incrémenter le compteur contenu dans le fichier chaque seconde.

Question 1.a. Dans un premier temps, nous allons traiter le cas dans lequel les arguments du script sont invalides. Les arguments sont invalides si (i) il n'y a pas un et un seul argument ou (ii) si l'unique argument est un chemin vers un répertoire existant (le test correspondant est **`[ -d chemin ]`**). Écrivez un script nommé **`cpt.sh`** qui renvoie faux (valeur 1) après avoir affiché un message d'erreur adéquat si les arguments sont invalides, et qui renvoie vrai (valeur 0) sinon.

Question 1.b. Dans le corps du script, écrivez la valeur 0 dans le fichier passé en argument (**`"$1"`**).

Question 1.c. Après avoir écrit 0 dans le fichier passé en argument, votre script doit maintenant effectuer une boucle infinie. Dans cette boucle, votre script doit, chaque seconde, incrémenter la valeur contenu dans le fichier. Pour cela, votre script doit :

- lire le contenu du fichier passé en argument dans une variable **`n`**,
- additionner 1 à **`n`** avec la commande **`expr`**,
- écrire **`n`** dans le fichier passé en argument,
- attendre une seconde en invoquant la commande **`sleep 1`**.

Remarque : *Une boucle infinie s'écrit de la façon suivante :*

```
while true; do ... done
```

Pour arrêter un script qui ne se termine jamais comme celui que vous mettez en œuvre à cette question, il suffit de saisir la combinaison de touches ***control+c****.*

Exercice 2 : Découper des lignes (~40mn)

Le but de cet exercice est d'écrire un petit programme capable de découper des lignes, c'est-à-dire d'extraire certains mots séparés par des blancs.

Question 2.a. Pour commencer, nous allons vérifier que les arguments du script ne sont pas invalides. Les arguments sont invalides si :

- le nombre d'arguments n'est pas égal à 2,
- ou si le premier argument ne correspond pas à un fichier existant (test **`[ ! -e fic ]`**),
- ou si le premier argument correspond à un répertoire (test **`[ -d fic ]`**).

En cas d'erreur, le script doit quitter en affichant un message adéquat et en renvoyant faux (valeur 1), sinon, il doit quitter en renvoyant vrai (valeur 0).

Question 2.b. Complétez votre script pour qu'il lise, ligne à ligne, le fichier passé en argument (premier argument) et écrive ces lignes sur la sortie standard. On vous rappelle que pour lire un fichier ligne à ligne à partir du fichier **`fichier`**, il faut utiliser la construction **`while read line; do ... done <fichier`**.

Question 2.c. Modifiez votre script pour qu'il itère sur chaque mot de chaque ligne non vide (indiqué par **`[ -n "$line" ]`**), et affiche le mot sur une ligne séparée. Pensez à utiliser une boucle **`for`** pour itérer sur les mots de la ligne précédemment lue.

Question 2.d. Modifiez votre script pour qu'il n'affiche que le $\mathbf{n}^{\text{ième}}$ mot de chaque ligne, où **`n`** est le second argument du script.

Pour mettre en œuvre cet algorithme, nous vous conseillons d'utiliser une variable annexe, par exemple **`num`**, que vous initialiserez à zéro avant d'itérer sur les mots, et que vous incrémenterez à l'aide de la commande **`expr`** à chaque itération de la boucle sur les mots.

Remarque : *L'algorithme proposé ne traite pas le cas dans lequel une ligne contient moins de* ***n*** *mots. Dans ce cas, l'algorithme va simplement ignoré la ligne et ne rien afficher. Vous pouvez, en hors présentiel, traiter ce cas et afficher une ligne vide lorsqu'une ligne contient moins de* ***n*** *mots.*

Exercice 3 : Duplication de flux (~30mn)

Le but de cet exercice est d'écrire un script qui prend en argument un fichier. Votre script doit lire les lignes provenant de l'entrée standard, et les écrire à la fois dans le fichier passé en argument et sur la sortie standard.

Remarque : *Vous n'aurez probablement pas le temps de finir cet exercice pendant la séance. Il vous ait demandé de le terminer en hors présentiel.*

Question 3.a. Dans un premier temps, nous allons traiter le cas où les arguments du script sont invalides. Les arguments sont invalides si (i) il n'y a pas un et un seul argument ou (ii) si l'unique argument est un chemin vers un répertoire existant (le test correspondant est **`[ -d chemin ]`**). Écrivez un script nommé **`tee.sh`** qui renvoie faux (valeur 1) après avoir affiché un message d'erreur adéquat si les arguments sont invalides, et qui renvoie vrai (valeur 0) sinon.

Question 3.b. Modifiez votre script de façon (i) à lire l'entrée standard ligne à ligne et (ii) à afficher chaque ligne lue sur la sortie standard. On vous rappelle que pour lire l'entrée standard ligne à ligne, il faut utiliser la construction suivante : **`while read line; do ...; done`**.

Remarque : *Vous pouvez tester votre script de façon interactive en écrivant des lignes sur le terminal. Dans ce cas, vous pouvez fermer le flux associé à l'entrée standard avec la combinaison de touches* ***control-d****. Vous pouvez aussi tester votre*

script en redirigeant l'entrée standard à partir d'un fichier, par exemple avec ***`./tee.sh fic < /etc/passwd`****.*

Question 3.c. Nous allons maintenant dupliquer la sortie vers le fichier passé en argument. Ouvrez un flux associé au fichier passé en argument au début du corps du script. Vous devez ouvrir le flux en écriture et en écrasement. (commande **`exec 3>fichier`**).

Modifiez aussi votre boucle de façon à écrire chaque ligne lue à la fois sur la sortie standard (commande **`echo ligne`**) et dans le flux ouvert (commande **`echo ligne >&3`**).

Exercice 4 : Configuration de votre compte (~30mn – optionnel mais fortement conseillé)

Objectifs : *Mettre en place une configuration par défaut pour* ***`bash`****. Cet exercice, bien que non obligatoire, est fortement conseillé car il vous permet de personnaliser votre* ***`bash`****, ce qui, à l'usage, est très commode.*

Remarque : *Pour les utilisateurs de* ***`MacOS`****, commencez par exécuter la commande suivante :* ***`ln -s ~.bashrc ~/.bash_profile`***

Question 4.a. Invite de commande Un invite de commande, ou *prompt*, correctement positionné est un atout majeur pour se repérer dans le système de fichiers sans avoir à utiliser les commandes spécifiques. Nous vous proposons dans cet exercice de personnaliser votre invite de commande.

1. Affichez la valeur actuelle de la variable d'environnement correspondant au prompt.
2. Accédez à la page de manuel de bash, section « Invite » (ou « Printing-a-Prompt », ou « PROMPTING » si la documentation est en anglais), afin de consulter les caractères spéciaux d'encodage de l'invite de commande.
3. Faites en sorte que votre invite de commande s'affiche comme suit :
   `[votre-login@nom-de-votre-machine dernier-élément-du-répertoire-courant]$ `␣, où « ␣ » représente un espace.

   Par exemple, si vous êtes dans le répertoire **`~/cours/csc3102/bulbizarre/`**, que vous vous appelez **`sacha`** et que vous vous trouvez sur la machine **`argenta.imtbs-tsp.eu`**, votre invite de commande doit s'afficher comme suit : **`sacha@argenta.imtbs-tsp.eu:bulbizarre$ `**␣
4. On vous rappelle que pour pérenniser votre configuration, vous devez ajouter la commande de personnalisation de l'invite de commande dans le fichier **`~/.bashrc`**. Ouvrez le fichier nommé **`~/.bashrc`** avec votre éditeur de texte et copiez y la commande précédente. On vous rappelle que la procédure pour copier/coller du texte avec la souris sous GNU/Linux est présentée ici.

Question 4.b. Création d'un alias de commande

1. Définissez l'alias **`rm`** comme étant la commande **`rm`** avec l'option qui demande à l'utilisateur de confirmer l'effacement de chaque fichier. Faîtes de même avec les commandes **`cp`** et **`mv`**.
2. Créez un fichier nommé **`test.txt`** avec la commande **`touch test.txt`**.
3. Supprimez le fichier **`test.txt`** avec la commande **`rm`** et constatez qu'une confirmation de suppression vous est bien demandée.
4. Pérennisez les alias des commandes **`rm`**, **`cp`** et **`mv`** en les copiants/collants dans le fichier **`~/.bashrc`**.

Question 4.c. Droit d'accès par défaut

1. Quelle est la commande permettant de connaître les droits d'accès positionnés sur les fichiers au moment de leur création ?
2. Quels sont les droits d'accès par défaut configurés sur votre compte ?
   Positionnez le masque de manière à supprimer les droits en lecture et en écriture pour le groupe et les autres. (Le droit d'exécution est par défaut oté sur les fichiers mais activé pour les répertoires pour en autoriser la traversée.)
3. Vérifiez que cette commande a fonctionné en créant un nouveau fichier (commande **`touch`**) et un nouveau répertoire (commande **`mkdir`**) pour en vérifier les droits d'accès.
4. Pérennisez votre masque en copiant/collant votre commande **`umask`** dans le fichier **`~/.bashrc`**.

Exercice 5 : Lignes alternées (~40mn – défi)

Le but de cet exercice est d'écrire les lignes provenant de deux fichiers de façon alternée sur la sortie standard.

Question 5.a. Dans un premier temps, nous allons traiter le cas où les arguments du script sont invalides. Les arguments sont invalides si (i) il n'y a pas deux arguments ou (ii) si l'un des deux arguments n'est pas chemin vers un fichier existant (le test correspondant est **`[ -f chemin ]`**). Écrivez un script nommé **`paste.sh`** qui renvoie faux (valeur 1) après avoir affiché un message d'erreur adéquat si les arguments sont invalides, et qui renvoie vrai (valeur 0) sinon.

Question 5.b. Votre script doit maintenant ouvrir deux flux en lecture : l'un associé au fichier passé en premier argument et l'autre associé au fichier passé en second argument.

Question 5.c. Pour simplifier, nous supposons pour le moment que les deux fichiers ont le même nombre de lignes. Affichez chacune des lignes des deux fichiers en les alternant. Pour vous guider, vous devez écrire une boucle qui affiche une ligne de chaque fichier, et ceci tant qu'il reste des lignes à lire dans le premier fichier et dans le second. L'algorithme est donc le suivant :

```
Tant que lecture premier flux renvoie vrai et (opérateur &&) lecture second flux renvoie vrai
  Écrit la ligne lue à partir du premier flux sur la sortie standard
  Écrit la ligne lue à partir du second flux sur la sortie standard
```

Testez votre script avec :

```
$ ./paste.sh ./paste.sh ./paste.sh
```

Question 5.d. Nous allons maintenant gérer des fichiers de tailles différentes. Il faut commencer par transformer votre boucle en boucle infinie (on vous rappelle qu'une boucle infinie s'écrit **`while true; do ... done`**).

Dans votre boucle, commencez par lire une ligne du premier fichier :

- si la lecture dans le premier fichier renvoie vrai, lisez une ligne du second fichier
    - si la lecture dans le second fichier renvoie vrai, affichez les lignes des premier et second fichiers
    - sinon afficher la ligne du premier fichier
- sinon (c.-à-d., la lecture sur le premier fichier renvoie faux) lisez une ligne du second fichier
    - si la lecture dans le second fichier renvoie vrai, affichez la ligne du second fichier
    - sinon quittez votre script avec un **`exit 0`**

Testez votre script avec :

```
$ ./paste.sh ./paste.sh /etc/passwd
```

TP4 – Chasse au trésor

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs : *Trouver un trésor et, accessoirement, réviser toutes les notions de CSC3102 vues jusqu'à présent.*

Voici l'ultime trace du trésor confié à Bilbon lors de l'un de ses longs voyages itinérant de la fin du XX^{è} siècle.

« Face à la clé du trésor cachée dans les itinéraires signés de mes nombreux voyages à travers de vastes pays de la fin du siècle dernier, l'anagramme des premières lettres des trois premières lignes selon l'ordre alphabétique portant le mot "à" en garantira l'authenticité. La clé ainsi découverte, il vous faudra la réorganiser selon un ordre lexicographique de troisième champ afin d'en extraire les 3^{è} mots des couples de lignes de tête et de queue de lignes, et ainsi découvrir le chemin jusqu'au trésor ! »

Exercice 1 : Mise en place de la chasse au trésor (~5mn)

Question 1.a. Récupérez l'archive des carnets de voyage de Bilbon en utilisant la commande **`wget`**.

```
$ wget http://www-inf.it-sudparis.eu/COURS/CSC3102/Supports/ci4-outils/exo-chasse-au-tresor/CarnetsDeVoyage.tar.gz
```

Question 1.b. Extrayez l'archive.

Exercice 2 : Identification du voyage (~1h10)

Objectifs : *Identifier le répertoire dans lequel se trouve la clé du trésor.*

La clé du trésor est cachée au sein d'un des répertoires de l'archive extraite. Chacun des répertoires correspond à un voyage, dont chacun des noms a été construit sur le même modèle : **`<année>-<lieu>`**, associé aux date et destination du voyage entrepris. Le champ **année** est toujours constitué de quatre chiffres. Quant au champ **`lieu`**, il correspond soit à un pays, et dans ce cas il commence par une majuscule, soit à une ville et il commence par une minuscule. Les deux champs sont séparés soit par un tiret haut, soit par un tiret bas.

La clé du trésor ayant été cachée lors d'un voyage itinérant à la fin du siècle dernier, nous sommes à la recherche d'un répertoire dont la date de création est comprise entre 1970 et 1999, et dont le lieu référence un pays. Par exemple, 1986-Bolivie.

Afin de mener à bien la recherche de la clé du trésor, nous vous proposons de procéder de manière incrémentale. Pour cela, vous allez écrire un script nommé **`chasse.sh`** que vous modifierez à chaque étape.

Remarque : *À chaque étape, pensez à vérifier que votre script est correct en l'exécutant.*

Question 2.a. Commencez par écrire le script nommé **`chasse.sh`** dans le répertoire courant (c.-à-d., le répertoire parent de **`CarnetsDeVoyage`**).

Ce script doit définir une variable nommée **`base`**, et lui affecter le chemin **`CarnetsDeVoyage`**.

Lancez votre script et vérifiez que votre script est correct en affichant cette variable à l'aide d'un **`echo`**.

Remarque : *Dans toute la suite de l'exercice, nous supposons que votre répertoire courant est le répertoire parent de* ***`CarenetsDeVoyage`****.*

Question 2.b. Votre script doit maintenant sélectionner tous les fichiers qui correspondent à des voyages. En procédant étape par étape et en utilisant des motifs de filtrage **bash**, modifiez la commande **echo** de la question précédente pour afficher :

- tous les fichiers dont le nom commence par 19 se trouvant dans le répertoire **$base**,
- tous les fichiers dont le nom commence par 19 et est suivi par un chiffre entre 7 et 9,
- tous les fichiers dont le nom commence par 19, est suivi par un chiffre entre 7 et 9, et est suivi d'un chiffre quelconque,
- tous les fichiers dont le nom commence par 19, est suivi par un chiffre entre 7 et 9, est suivi d'un chiffre quelconque, et est suivi soit par un tiret bas soit par un tiret haut.
- tous les fichiers dont le nom commence par 19, est suivi par un chiffre entre 7 et 9, est suivi d'un chiffre quelconque, est suivi soit par un tiret bas soit par un tiret haut, et est suivi d'une chaîne de caractères quelconque dont la première lettre est une majuscule.

Pour vous aider, vous devriez écrire une commande du type

```
echo $base/[un-motif-ici]
```

Remarque : *Vérifiez manuellement que les entrées sélectionnées correspondent bien aux descriptions données.*

Question 2.c. À ce stade, il doit vous rester 4 entrées candidates car il n'y a eu aucun filtre sur le type des entrées considérées.

Nous sommes à la recherche d'un plan d'itinéraire. Nous pouvons donc raffiner notre quête en nous limitant aux répertoires dans lequel le trésor aura possiblement été caché. Avant de vérifier si un fichier est un répertoire, votre script doit d'abord itérer sur les noms trouvés à la question précédente.

Au lieu d'afficher les noms trouvés avec un **echo**, votre script doit donc itérer sur les noms trouvés avec un **for**, et les afficher chacun sur une ligne. Le schéma d'algorithme est le suivant :

```
for x in expression-régulière-question-précédente; do
  echo $x
done
```

Question 2.d. Modifiez le corps de la boucle de façon à n'afficher que les répertoires.

Question 2.e. Le voyage ayant été long, votre script doit sélectionner le répertoire le plus volumineux. Vous devez donc modifier votre boucle de façon à stocker, dans une nouvelle variable nommé **rep**, le répertoire le plus volumineux. Nous procédons en plusieurs étapes. Dans un premier temps, remplacez le **echo** de la commande précédente de façon à afficher la taille de chacun des répertoires. Pour ne pas afficher la taille des sous-répertoires, pensez à utiliser l'option **-d0** de **du**.

Question 2.f. En utilisant un tube et la commande **cut**, n'affichez que la taille des répertoires (c.-à-d., sans leur nom).

Question 2.g. Stockez cette taille dans une variable que vous appellerez **cur**, puis affichez **cur** avec **echo**. Pensez à utiliser une imbrication de commandes, c.-à-d., la construction **$(...)**. Votre affichage doit être similaire à celui de la question précédente.

Question 2.h. Stockez dans la variable nommée **rep** le répertoire le plus volumineux et affichez le après la boucle. Pour cela, nous vous proposons d'utiliser l'algorithme suivant capable de trouver le répertoire le plus volumineux :

```
size prend la valeur 0

Pour tout x correspondant au motif
  Si x est un répertoire
```

```
    cur prend la taille du répertoire x
    Si size plus petit que cur
      size prend la valeur de cur
      rep prend la valeur de x
    FinSi
  FinSi
FinPour

Affiche rep
```

Remarque : *Si la variable* ***rep*** *est égale à la chaîne de caractères vide, c'est que quelque chose s'est mal passé pendant votre chasse.*

Bravo ! Vous avez découvert le voyage qui héberge la clé du trésor !

Exercice 3 : Identification de la clé du trésor (~30mn)

Objectifs : *Trouver le fichier qui sert de clé au trésor !*

Question 3.a. Pour identifier le fichier qui sert de clé, nous procédons étape par étape. Commencez par étendre le script **`chasse.sh`** pour afficher tous les fichiers ordinaires se trouvant dans le répertoire **`$rep`** ou dans un de ses sous-répertoires, et dont le nom contient **`"Itineraire"`**.

Attention : *À cette question, il ne faut pas utiliser de recherche par motif comme vous l'avez fait au début de l'exercice 2. En effet, une recherche par motif cherche des fichiers dans un répertoire mais pas dans les sous-répertoires. À cette question, il faut donc utiliser la commande* ***`find`*** *qui permet de chercher des fichiers dans un répertoire et dans ses sous-répertoires. On vous rappelle que pour trouver des fichiers ordinaires, il faut utiliser l'option* ***`-type f`*** *de* ***`find`****.*

Question 3.b. Comme nous devons accéder au contenu de chacun des fichiers trouvés, nous devons itérer sur les fichiers trouvés à la question précédente. Commencez par faire une boucle **`for`** pour itérer, avec une variable **`x`**, sur chacun de ces fichiers (pensez à utiliser la construction **`$(...)`** pour récupérer la sortie produite à la question précédente). Dans le corps de cette boucle **`for`**, affichez le nom du fichier sur une ligne avec la commande **`echo`**. L'affichage produit doit être similaire à celui de la question précédente.

Question 3.c. Dans le corps de la boucle, au lieu d'afficher le nom du fichier, cherchez dans le fichier la signature de son auteur, c.-à-d. « **`Bilbon`** », avec la commande **`grep`**.

Question 3.d. Au lieu d'afficher la ligne correspondant au motif « **`Bilbon`** », affichez le nom du seul fichier ayant pour signature **`Bilbon`**. Pour vous guider, il faut savoir que la commande **`grep`** renvoie **`vrai`** si elle trouve une ligne qui correspond au motif. Comme vu dans le cours sur les flux, rappelez-vous aussi que vous pouvez éliminer l'affichage de la commande **`grep`** en redirigeant sa sortie vers le fichier **`/dev/null`**. En d'autres termes, vous pouvez utiliser un code similaire à celui-ci :

```
if grep motif fichier >/dev/null; then ... fi
```

Question 3.e. Le but de cette question est de résoudre une première partie de l'énigme : « l'anagramme des premières lettres des trois premières lignes selon l'ordre alphabétique portant le mot "à" en garantira l'authenticité ». Au lieu d'afficher le fichier trouvé comme à la question précédente, utilisez des tubes pour chaîner les commandes suivantes :

- affichez, avec la commande **`grep`**, toutes les lignes contenant le caractère « **à** » dans le fichier,
- triez, avec la commande **`sort`**, ces lignes en suivant l'ordre lexicographique,
- ne gardez que les trois premières lignes du fichier avec la commande **`head`**,
- ne gardez que la première lettre de chaque ligne avec la commande **`cut`**,

- supprimez les retours à la ligne avec la commande **tr** (caractère « **\n** »);

Remarque : *Nous vous conseillons vivement de procéder étape par étape.*

Remarque : *Si Bilbon ne se moque pas de vous, vous devriez obtenir un anagramme du mot « CLE ». Remarquez que, comme vous avez trié les lignes en suivant l'ordre lexicographique, cet anagramme ne peut être que « CEL ».*

Question 3.f. Lorsque vous trouvez le fichier contenant l'anagramme du mot « CLE », vous devez stocker le nom de ce fichier dans une variable nommée **cle**. Pour vous assurer que votre script est correct, affichez la valeur de cette variable à la fin du script. Si votre script est correct, il devrait vous afficher le fichier contenant l'anagramme du mot « CLE ».

Bravo ! Vous avez identifié le fichier contenant la clé !

Exercice 4 : Ultime étape : la découverte du trésor ! (~45mn)

Maintenant que vous avez identifié le fichier contenant la clé du trésor, vous allez (enfin !) pouvoir trouver le trésor.
Nous vous rappelons que la fin de l'énigme est : *« La clé ainsi découverte, il vous faudra la réorganiser selon un ordre lexicographique de troisième champ afin d'en extraire les 3è mots des couples de lignes de tête et de queue de lignes, et ainsi découvrir le chemin jusqu'au trésor ! »*

Question 4.a. Nous commencons par trier le fichier stocké dans la variable **cle** *selon un ordre lexicographique de troisième champ*. Dans le script **chasse.sh**, remplacez l'affichage de la variable **cle** par l'affichage du fichier **cle**, trié selon le troisième champs de chaque ligne (option **-k3,3** de **sort**).

Remarque : *Par défaut, la commande **sort** de Linux ignore la casse, mais ce n'est pas le cas sur un MacOS. Pour cette raison, les utilisateurs de MacOS doivent ignorer la casse en utilisant l'option **-f**.*

Question 4.b. *« [...] extraire les 3e mots des couples de lignes de tête et de queue de lignes [...] »*. Pour réussir à résoudre cette partie de l'énigme, nous vous proposons de :

- éliminez les lignes vides ,
  - Filtrez le résultat de la commande précédente de façon à ne conserver que les lignes qui contiennent du texte (sachez que le motif **[[:alpha:]]** de **grep**, comme celui de **bash**, permet de capturer n'importe quel caractère).
  - Sauvegardez ce résultat dans un fichier nommé **Itineraire_trie.txt**.
  - Isolez les deux premières lignes du fichier **Itineraire_trie.txt** et sauvegardez-les dans un fichier nommé **Itineraire_trie_compact.txt**.
  - Isolez les deux dernières lignes du fichier **Itineraire_trie.txt** et ajoutez-les au fichier nommé **Itineraire_trie_compact.txt**.
  - Stockez dans une variable nommée **mots** les troisièmes mots de chacune des lignes de **Itineraire_trie_compact.txt**.
  - Supprimez les fichiers **Itineraire_trie.txt** et **Itineraire_trie_compact.txt**.
  - Enfin, affichez le contenu de la variable **mots**.

Remarque : *Nous vous conseillons de procéder étape par étape et de vérifier, à chaque étape, que votre code fonctionne correctement.*

Question 4.c. Afin d'afficher le trésor, il faut d'abord reconstituer le chemin vers le trésor dans une variable nommée **tresor**. Pour cela, il faut ajouter des « / » entre les différents mots de la variable **mots**. Au lieu d'afficher la variable **mots**, nous vous proposons donc l'algorithme suivant :

```
tresor prend la valeur de la variable base
Pour tout mot dans mots
  ajouter '/' et mot à la fin de tresor
FinPour
```

```
Afficher la variable tresor
```

Question 4.d. Ultime étape ! Au lieu d'afficher le chemin du trésor, affichez son contenu.

Bravo ! Vous avez découvert le trésor !

TP5 – Processus

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs :

- *Savoir observer les processus s'exécutant sur une machine*
- *Manipuler un processus en cours d'exécution*
- *Savoir tuer un (ensemble de) processus*
- *Notions de concurrence*

Exercice 1 : Observer un processus (~15mn)

Question 1.a. Trouvez, à l'aide de la commande **`ps`**, le PID du processus **`dhclient`**.

Question 1.b. Ajoutez l'option **`-l`** à la commande précédente et trouvez le nom du processus parent du processus dhclient.

Question 1.c. Utilisez maintenant la commande **`pstree`** afin de trouver le PID du processus **`dhclient`** ainsi que le nom de son processus père.

Question 1.d. Le répertoire **`/proc/$PID`** contient des fichiers décrivant l'état du processus **`<PID>`**. Trouvez le fichier contenant la ligne de commande ayant servi à lancer le processus **`dhclient`**.

Question 1.e. Consultez la documentation de la commande **`yes`**. À quoi sert-elle ?

Question 1.f. Ouvrez un terminal et lancez la commande suivante :

```
$ yes "Unix, c'est super"
```

Pendant l'exécution de la commande **`yes`**, lancez la commande **`top`** dans un autre terminal. Quels sont les processus qui consomment le plus de CPU ?

Exercice 2 : Suspendre un processus (~15mn)

Question 2.a. Lancement en avant-plan

1. Tapez la commande «**`emacs`**» pour lancer, en avant-plan, le processus **`emacs`**
2. Pour quelle raison l'invite de commande (prompt) n'est plus présente dans le terminal ? Tapez la commande **`ls`** dans le terminal et observez le résultat.
3. Dans le terminal, tapez **`control+z`** pour stopper le processus en avant-plan (sans le détruire) et ainsi pouvoir utiliser le terminal.
4. Dans le terminal, vérifiez que vous pouvez exécuter la commande **`ls`**. Dans la fenêtre **`emacs`**, tapez quelques caractères pour vérifier son fonctionnement.
5. Lancez la commande **`ps -l`** pour identifier le processus qui s'exécute et connaître l'état des autre processus lancés depuis ce terminal. Consultez la documentation de la commande **`ps`** pour trouver cette information.
6. Tapez la commande **`fg`** pour relancer le processus **`emacs`** en *avant-plan*. Pouvez vous maintenant écrire dans la fenêtre emacs ? Dans le terminal ?

Question 2.b. Lancement en arrière-plan

1. Si ce n'est pas déjà fait, tapez la commande «**`emacs`**» pour lancer, en avant-plan, le processus **`emacs`**
2. Dans le terminal, tapez **`control+z`** afin de suspendre le processus **`emacs`**. Vérifiez avec **`ps`** que le processus **`emacs`** est bien suspendu.

3. Tapez la commande **bg** pour relancer le processus **emacs** en *arrière-plan*. Pouvez-vous maintenant écrire dans la fenêtre **emacs** ? Dans le terminal ? Quelle est la différence entre les commandes **fg** et **bg** ?
4. Utilisez la commande **ps** pour afficher l'état du processus **emacs**.
5. Fermez le programme emacs, puis relancez-le en terminant la ligne de commande par & : **emacs &**. Vérifiez que le terminal reste utilisable. Vérifier avec **ps** que l'état du processus **emacs** est le même qu'à la question précédente.
6. Lors du lancement du processus emacs à la question précédente, un nombre est apparu sur le terminal. À quoi correspond-il ?

Exercice 3 : Tuer des processus (~15mn)

Question 3.a. Tuer un processus

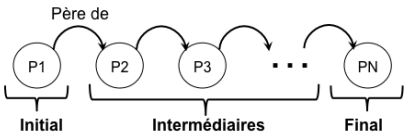
1. Lancez 2 processus **xeyes** en arrière-plan.
2. Retrouvez le PID du premier processus xeyes lancé et utilisez la commande kill pour le détruire.
3. Vérifiez que le processus n'existe plus.

Question 3.b. Tuer plusieurs processus

1. Relancez un processus **xeyes** en arrière-plan et vérifiez qu'il existe bien 2 processus **xeyes**.
2. Utilisez la commande **killall** pour terminer tous les processus **xeyes** en cours d'exécution.

Exercice 4 : Chaîne de processus (~1h15)

Cet exercice a pour but de vous apprendre à créer plusieurs processus en utilisant un algorithme récursif. Comme présenté sur la figure qui suit, vous allez créer une chaîne de **N** processus. Dans une chaîne de processus, chaque processus est le père du suivant, sauf pour le dernier processus de la chaîne, qui lui, n'a pas de fils. Le premier processus de la chaîne est appelé le processus initial et le dernier le processus final. Les autres processus sont appelés les processus intermédiaires.



L'algorithme que nous utilisons est récursif. Si **N** vaut 1, alors l'unique processus de la chaîne est le processus final. Il ne doit donc rien faire de spécial. Si **N** est strictement supérieur à 1, alors le processus doit encore créer **N-1** processus. Pour cela, il lui suffit de lancer **chaine.sh** avec comme paramètre **N-1**.

Question 4.a. Nous allons écrire le script qui crée une chaîne de processus étape par étape. Pour commencer, écrivez un script nommé **chaine.sh** qui :

- Vérifie ses arguments. Si le script ne reçoit pas un unique argument, il doit quitter avec un message d'erreur et un code de retour faux.
- Affiche le message « **Processus X démarre** », où **X** est le **PID** du processus.
- Quitte avec un code de retour vrai.

Question 4.b. Avant de créer récursivement les processus, dans cette question, nous affichons ce que l'algorithme est supposé faire. Modifier votre script de façon à ce que :

- si **N** est strictement supérieur à 1, votre script affiche « **Il reste K processus à créer** », où **K** est égal à **N-1**,
- sinon, votre script affiche « **Fin de chaîne** ».

Question 4.c. Nous finalisons maintenant l'algorithme. Après avoir affiché « **Il reste K processus à créer** », où **K=N-1**, votre script doit donc lancer la commande **./chaine.sh K** pour créer le processus suivant de la chaîne.

Vérifiez que vous démarrez bien 4 processus lorsque vous invoquez **./chaine.sh 4**.

Question 4.d. Il faut maintenant lancer les processus en tâche de fond. Un processus ne doit aussi se terminer que lorsque son fils direct est terminé. Enfin, tous les processus (y compris le processus final) doivent afficher **Processus X termine**,

où **X** est le **PID** du processus courant. Pour les processus qui possèdent un fils, cet affichage doit avoir lieu après avoir attendu la fin du fils.

Modifiez votre script en conséquence. Vérifiez aussi que les messages de terminaison s'affichent dans l'ordre inverse des créations en lançant **`./script.sh 4`**.

Question 4.e. Nous souhaitons maintenant communiquer le **PID** du processus initial jusqu'au processus final. Le langage **bash** fournit un mécanisme pour connaître le **PID** de son parent direct, mais pas de ses autres aïeuls. Il faut donc communiquer ce PID via un nouveau mécanisme. Pour mettre en œuvre un tel mécanisme, nous devons résoudre deux sous-problèmes :

- Il faut être capable de communiquer le PID du processus initial jusqu'au processus final. Pour cela, nous utilisons une variable nommée **`pidInitial`**, initialisée par le processus initial à son PID, et exportée jusqu'au processus final.
- Il faut que le processus initial soit capable de se reconnaître de façon à initialiser et exporter cette variable **`pidInitial`**. Tous les processus, sauf le processus initial, vont démarrer avec une copie de la variable **`pidInitial`** puisque cette dernière est exportée par le processus initial. Le processus initial est donc l'unique processus pour lequel la variable **`pidInitial`** n'est pas définie initialement, c.-à-d., pour lequel la variable **`pidInitial`** a pour valeur la chaîne de caractère vide, qui se reconnaît avec la construction **`[ -z "$pidInitial" ]`**).

Modifiez votre script de façon à ce que le processus initial initialise et exporte la variable **`pidInitial`**. Pour vous assurer que votre programme est correct, modifiez aussi l'affichage « **`Processus X démarre`** », où **X** est le **PID** du processus courant, en « **`Processus X démarre avec le processus initial Y`** », où **Y** est le **PID** du processus initial. Testez votre programme en lançant **`./chaine.sh 4`**.

Félicitation ! Vous venez d'écrire votre première application multi-processus complexe !
Mettez le script `chaine.sh` de côté, vous vous en servirez à la prochaine séance.

Exercice 5 : Chasse au trésor – (~30mn – hors présentiel)

Attention : Cet exercice nécessite un Linux et il n'est actuellement pas possible de le faire avec un Mac OS.

*De retour de vacances, Bilbon décide de reprendre ses notes concernant la généalogie du nain Gimli. Au milieu de ses dossiers, il retrouve le programme **`genealogie`** qui, d'après ses souvenirs, permet de modéliser l'arbre généalogique du célèbre nain.*

Question 5.a. Mise en place de la chasse au trésor

1. Récupérez l'archive contenant les notes de Bilbon mise à votre disposition sur Moodle ou ici.
2. Extrayez l'archive dans votre arborescence
3. Déplacez vous dans le répertoire ainsi créé et lancez la commande **`make`**.

Question 5.b. La chasse au trésor

1. Lancez le programme genealogie.
2. *Vous êtes accueillis par Gimli qui va vous aider à mettre en ordre les notes de Bilbon sur sa généalogie. Dans un premier temps, aidez le à retrouver la fiche concernant son père.*

   Retrouvez le PPID du processus **`gimli`**.
3. *Gimli se rend compte d'une erreur dans la fiche de son père qui ne mentionne pas sa mort. Corrigez cela dans le programme de généalogie.*

   Tuez le processus père du processus **`gimli`**.
4. *Vous êtes interrompus dans vos recherche par Oin, l'oncle de Gimli, qui annonce l'heure du goûter. Mais un des nains de la famille se goinfre et est en train de manger toutes les tartes. Retrouvez celui qui s'empiffre.*

   Trouvez le PID du processus qui consomme beaucoup de CPU.

5. *Afin que les autres nains puissent participer au gouter, ils décident de punir le responsable en le mettant tout au bout de la table, loin des tartes.*

   Donnez la priorité **17** au processus glouton.
6. *Une fois le glouton puni, le gouter se déroule dans un calme relatif. Mais Gimli sonne la fin de la pause. Il est en effet l'heure de partir à la taverne. Mais les chevaux sont enfermés à clé dans l'écurie. La clé a été confiée à Balin qui a oublié où il l'avait rangé. Il a pourtant noté sur un bout de papier l'endroit où est rangée la clé…*

   Trouvez le deuxième paramètre qui a été passé au processus **`balin`** à son lancement.
7. *Les nains peuvent donc ouvrir l'écurie et se rendre à la taverne. Pendant que les nains festoient, Bilbon se désole du peu d'avancement de ses recherches…*

Remarque : *Pour les besoins de l'exercice, la généalogie exacte de Gimli a dû être altérée. Toute l'équipe enseignante prie les puristes du Seigneur des anneaux de l'en excuser.*

Exercice 6 : Concurrence (~30mn – défi)

Question 6.a. Le programme **`calcul.c`** effectue un calcul parfaitement inutile, mais qui a le mérite d'être très régulier (le programme effectue un nombre d'instructions fixe d'une exécution à l'autre pour une même entrée). Téléchargez ce programme en cliquant ici.

Pour compiler le programme, lancer la commande « **`gcc -o calcul calcul.c`** ».

Une fois le programme compilé, vérifiez la présence du fichier exécutable **`calcul`** dans votre répertoire et lancez le programme **`calcul`** avec le paramètre 60000. Observez le temps nécessaire au calcul.

Question 6.b. Les machines de la salle de TP sont équipées de processeurs *multicœurs*. Cela veut dire que la machine contient plusieurs « processeurs » et peut donc exécuter plusieurs programmes simultanément.

Utilisez la commande **`lscpu`** pour trouver le nombre de cœurs de votre machine (voir la ligne commençant par « **Proc** »).

Question 6.c. Ecrivez un script **`calcul_parallele.sh`** qui exécute **`N`** fois la commande **`./calcul 60000`** en arrière-plan.

Question 6.d. Exécutez le script **`calcul_parallele.sh`** et observez le temps d'exécution des différents processus. La variation d'un processus à l'autre est-elle grande ?

Question 6.e. Modifiez maintenant le script **`calcul_parallele.sh`** pour qu'il exécute **`N+1`** fois la commande **`./calcul 60000`** en arrière-plan. Exécutez le script. Que remarquez vous ?

Question 6.f. Modifiez le script **`calcul_parallele.sh`** afin que l'un des processus soit lancés avec la priorité **`19`**.

Question 6.g. Exécutez le script **`calcul_parallele.sh`**. Que pouvez vous conclure ?

Question 6.h. Modifiez le script **`calcul_parallele.sh`** de façon à démarrer N+1 processus de calcul avec la priorité 0, puis à changer, au bout de 3 secondes, la priorité de l'un des processus. Exécutez le script et vérifiez que vous avez bien le comportement attendu.

Remarque : *Vous aurez besoin de connaître le PID d'un des processus fils pour changer sa priorité. Pour cela, vous devez savoir que la variable **`$!`** est égale au PID du dernier processus lancé. Par exemple:*

```
$ xeyes &
[1] 9645
$ echo $!
9645
$
```

TP6.1 – Les signaux

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs :

- *Manipuler les signaux (**`trap`**, **`kill`**)*
- *Réviser les redirections*

Exercice 1 : *Ave César* (~1h30)

Le but de cet exercice est de vous faire manipuler les signaux. Pour cela, nous concevons un (tout petit) Colisée de Rome à partir de la chaîne de processus que vous avez mise en œuvre à la séance précédente (voir ici). Le Colisée est constitué de gladiateurs qui saluent César, et de César qui les trucide dans la joie.

Partie 1 : Première partie : les gladiateurs

Dans cette première partie, nous créons les gladiateurs. Un gladiateur est un processus qui affiche toutes les 5 secondes « **`X: Ave César`** », où **`X`** est le **`PID`** du gladiateur, et qui meurt en criant « **`X: Morituri te salutant`** » lorsqu'il reçoit un signal **`USR1`**.

Question 1.a. Copiez le script **`chaine.sh`** de l'exercice sur les chaînes de processus en **`gladiateur.sh`**. Vous pouvez soit partir de votre solution, soit télécharger le fichier se trouvant **`ici`**.

Dans votre script, au lieu d'attendre la mort d'un enfant, d'afficher « Processus X termine » et de retourner un code d'erreur vrai, **`gladiateur.sh`** doit maintenant exécuter une boucle infinie (**`while true; do ... done`**) qui affiche toutes les 5 secondes « **`X: Ave César`** », où X est le **`PID`** du gladiateur.

Remarque : *Dans cette question, comme dans toute cette partie, vous devez systématiquement tester votre script en ne lançant qu'un unique gladiateur (**`gladiateur.sh 1`**). Rappelez-vous que vous pouvez interrompre votre script en saisissant la combinaison de touches **`control+c`**.*

Question 1.b. Complétez votre script pour qu'il affiche « **`X: Morituri te salutant`** » lorsqu'il reçoit un signal **`USR1`**. Testez votre script en lançant un gladiateur dans un terminal (**`./gladiateur.sh 1`**), et en envoyant un signal **`USR1`** au gladiateur avec la commande **`kill`** dans un autre terminal.

Question 1.c. Complétez votre script pour qu'il termine le processus en renvoyant un code de retour vrai (0) juste après avoir affiché « **`X: Morituri te salutant`** ».

Question 1.d. Vous avez dû remarquer que lorsque vous envoyez un signal **`USR1`**, votre processus ne reçoit le signal qu'après avoir terminé son attente de 5 secondes. Ce comportement est dû au fait que la commande interne **`sleep`** masque les signaux pendant qu'elle s'exécute.

De façon à éviter cette attente, nous utilisons le fait que la commande **`wait`**, elle, peut être interrompue par un signal. Au lieu de lancer la commande **`sleep`** en avant plan, lancez-la en arrière plan, et utilisez **`wait $!`** pour attendre la fin de la commande **`sleep`** (le **`$!`** permet de n'attendre que la fin de la dernière commande, c.-à-d., **`sleep`**, et non celle de tous les enfants). Testez votre programme avec un unique gladiateur et vérifiez que le processus n'attend plus la fin de la commande **`sleep`**.

Partie 2 : Deuxième partie : le problème de la terminaison

Cette seconde partie de l'exercice a pour but de vous montrer qu'il est difficile de terminer plusieurs gladiateurs.

Question 1.e. Lancez deux gladiateurs avec la commande **`./gladiateur.sh 2`**. Envoyez un signal **`USR1`** a un des gladiateurs. Que constatez vous ?

Question 1.f. Débarrassez-vous du second gladiateur.

Question 1.g. Lancez deux gladiateurs avec la commande **`./gladiateur.sh 2`**. Saisissez la combinaison de touches **`control-c`**. Que constatez-vous ?

Question 1.h. Le cas échéant, débarrassez-vous des gladiateurs qui seraient encore en train de s'exécuter.

Partie 3 : Troisième partie : retrouver les PIDs des gladiateurs

Comme vous avez pu le constater, il est difficile de terminer plusieurs gladiateurs car lorsque vous envoyez un signal, il n'est envoyé qu'à un unique gladiateur. Dans cette partie, nous résolvons le problème avec le processus César. César, qui s'ennuie souvent au Colisée, apprécie envoyer des **`USR1`** à tous les gladiateurs pour les tuer. Pour connaître les **`PIDs`** des gladiateurs, César lit un parchemin nommé **`arene.txt`**, dans lequel se trouve un PID de gladiateur par ligne.

Question 1.i. Un gladiateur doit maintenant enregistrer son **`PID`** dans le fichier **`arene.txt`** de façon à ce que César puisse le trouver. Modifiez le script **`./gladiateur.sh`** pour qu'il ajoute son **`PID`** au fichier **`arene.txt`** après avoir vérifié que les paramètres sont corrects. Testez votre script en lançant deux fois un unique gladiateur que vous interromprez avec un signal **`INT`**, et en vérifiant que **`arene.txt`** contient bien les **`PID`** des gladiateurs :

```
$ ./gladiateur.sh 1
Processus 2460 démarre avec le processus initial 2460
  Fin de chaîne
2460: Ave César
2460: Ave César
^C$ cat arene.txt
2460
$ ./gladiateur.sh 1
Processus 2473 démarre avec le processus initial 2473
  Fin de chaîne
2473: Ave César
^C$ cat arene.txt
2460
2473
```

Question 1.j. Avant d'envoyer des signaux, écrivez un script **`cesar.sh`** qui :

- affiche un message d'erreur et renvoie faux si **`arene.txt`** n'existe pas,
- lit ligne à ligne **`arene.txt`**, et affiche chaque ligne sur la sortie standard,
- supprime le fichier **`arene.txt`** après avoir lu chaque ligne,
- renvoie une code retour vrai.

Testez votre script avec le fichier **`arene.txt`** que vous avez généré à la question précédente.

Question 1.k. Au lien d'afficher les PIDs des gladiateurs, **`cesar.sh`** doit maintenant envoyer un **`USR1`** à chaque gladiateur enregistré dans **`arene.txt`**. Modifiez le script **`cesar.sh`** en conséquence.

Question 1.l. Nous pouvons maintenant utiliser le script **`cesar.sh`** pour terminer proprement tous les gladiateurs lorsque l'utilisateur saisie **`control-c`** ou utilise la commande **`kill`**. Modifiez **`gladiateur.sh`** de façon à lancer **`cesar.sh`** à la réception des signaux **`INT`** et **`TERM`**.

Félicitation ! Vous venez d'écrire votre premier protocole de terminaison !

Techniquement, le protocole que vous venez de mettre en œuvre permet de terminer proprement un ensemble de processus qui collaborent. Les gladiateurs sont des processus qui offrent un service à l'utilisateur (par exemple, chaque gladiateur pourrait être associé à un utilisateur connecté à un serveur Web), et César est le processus permettant de terminer proprement l'application.

TP6.2 – Les tubes

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

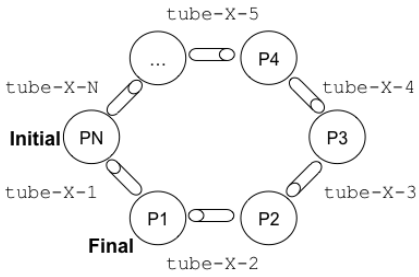
Objectifs :

- *Manipuler les tubes (**`mkfifo`**)*
- *Manipuler les signaux (**`kill`**, **`trap`**)*
- *Réviser les redirections*

Exercice 1 : La boîte à Meuh (~2h)

Le but de cet exercice est de transformer le Colisée que vous avez créé précédemment (voir ici) en boîte à Meuh, c'est-à-dire que les processus vont afficher **`Meuh`** les uns après les autres.

Même si afficher **`Meuh`** sur le terminal peut sembler quelque peu inutile, l'algorithme que vous allez mettre en œuvre est souvent utilisé dans les systèmes répartis et réseaux pour synchroniser un ensemble de processus. Par exemple, l'algorithme qu'on vous demande de mettre en œuvre est utilisé pour construire le protocole **`Token-Ring`** qui permet un accès équitable à une infrastructure réseau.



Techniquement, les processus sont organisés en anneau, comme présenté dans la figure ci-dessus. Chaque processus de la chaîne crée un tube nommé **`tube-X-N`**, où **`X`** est le **`PID`** du processus initial (stocké dans la variable **`pidInitial`**) et **`N`** l'argument donné au script, c.-à-d., le nombre de processus restant à créer dans la chaîne. Pendant l'exécution, les processus communiquent soit avec leur prédécesseur en accédant au tube du prédécesseur, soit avec leur successeur, en accédant à leur propre tube.

À partir de cette topologie, les processus échangent un jeton, symbolisé par le mot **`Meuh`**. Un processus reçoit le jeton lorsqu'il reçoit le mot **`Meuh`** via le tube de son prédécesseur. Le processus conserve ensuite le jeton pendant une seconde avant de le passer à son successeur en écrivant **`Meuh`** dans son tube.

Question 1.a. Pour commencer, il faut que chaque processus trouve (i) le nom du tube pour communiquer avec son prédécesseur et (ii) le nom du tube pour communiquer avec son successeur.

- Le nom du tube successeur est toujours **`tube-$pidInitial-N`**, où **`pidInitial`** est le **`PID`** du processus initial et **`N`** l'argument du script (i.e., **`$1`**).
- Pour le nom du tube prédécesseur :
  - si le processus est le processus initial (c.-à-d., le seul processus démarrant avec une variable **`pidInitial`** non initialisée), ce nom est **`tube-$$-1`**, où **`$$`** est le **`PID`** du processus courant, puisque le processus courant est le processus initial,
  - sinon, le nom du tube est **`tube-$pidInitial-M`**, où **`M`** vaut **`$1+1`**,

Après avoir copié **`gladiateur.sh`** en **`meuh.sh`**, modifiez votre script pour :

- stocker dans les variables **`pred`** et **`succ`** les noms des tubes du prédécesseur et du successeur,
- modifier l'affichage **`Processus $$ démarre avec le processus initial $pidInitial`** de façon à afficher les noms de ces tubes.

Remarque : *On ne vous demande pas de créer les tubes à cette question.*

Question 1.b. Modifiez votre script de façon à ce que chaque processus de la chaîne crée son tube successeur après avoir identifié les noms des tubes successeurs et prédécesseurs.

Question 1.c. Avant d'aller plus loin, il faut être capable de supprimer les tubes créés par notre application. Modifiez **`cesar.sh`** de façon à supprimer chacun des tubes créés. Vous pouvez remarquer que, comme **`meuh.sh`** exporte **`pidInitial`**, cette variable est positionnée dans **`cesar.sh`**. Pour cette raison, il suffit de supprimer tous les fichiers dont le nom commence par **`tube-$pidInitial-`** dans **`cesar.sh`**. Pensez à utiliser l'option **`-f`** de la commande **`rm`**, qui évite de demander une confirmation à l'utilisateur.

Question 1.d. Pour accéder aux tubes, nous utilisons des redirections avancées. En effet, des redirections simples ouvrent et ferment continuellement les tubes, ce qui entraîne des erreurs ou des blocages lorsque les tubes sont fermés pendant qu'il existe des interlocuteurs. Avant la boucle principale qui affiche **`Ave César`**, ouvrez en lecture/écriture les tubes prédécesseurs et successeurs (2 ouvertures différentes). Vous devriez remarquer que vous avez un message d'erreur du type : « mkfifo: tube-3665-1: File exists ». Nous nous occuperons de ce message à la question suivante.

Question 1.e. Le message d'erreur que vous avez est dû à un problème de synchronisation. Le processus initial arrive à atteindre l'ouverture du tube du prédécesseur avant que le processus final ait eu le temps créer son tube. Comme l'ouverture par le processus initial crée un fichier normal, la création du tube dans le processus final échoue avec un message d'erreur. Pour éviter ce problème, il ne faut donc ouvrir le tube du prédécesseur que si le tube existe. Avant d'ouvrir le tube du prédécesseur, ajoutez donc une boucle qui attend tant que le tube n'existe pas (donné par **`[ ! -e $pred ]`**) et qui dort une seconde à chaque pas de boucle. Vérifiez que le message d'erreur a bien disparu.

Question 1.f. De façon à mieux voir comment fonctionne le protocole, nous lançons chaque processus dans un terminal différent. Remplacez la création récursive d'un processus avec **`meuh.sh K`**, où **`K`** est égal au nombre de processus restant à créer dans la chaîne, par **`xterm -reverse -e meuh.sh K`** (l'option **`reverse`** crée un terminal blanc sur fond noir et est optionnelle, alors que l'option **`-e`** indique à **`xterm`** le nom du programme à exécuter dans le terminal). Vous pouvez admirer votre protocole de terminaison qui permet de fermer tous les terminaux en saisissant **`control-c`** dans n'importe quel terminal.

Question 1.g. Nous mettons maintenant en place notre boîte à Meuh. Dans la boucle principale, remplacez l'affichage de **`Ave César`** et l'attente de cinq secondes par :

- une lecture d'une ligne à partir du tube prédécesseur,
- l'affichage de la ligne lue,
- une attente d'une seconde,
- l'écriture de la ligne lue dans le tube successeur.

Pour amorcer votre boîte à Meuh, écrivez « **`Meuh`** » dans un des tubes à partir d'un autre terminal.

Question 1.h. Amorcer la boîte à Meuh à partir d'un autre terminal est relativement fastidieux. Pour cette raison, c'est le processus final qui va générer le premier jeton. Modifiez votre script pour que le processus final (celui qui a pour paramètre 1) écrive **`Meuh`** dans le tube de son successeur juste avant d'exécuter la boucle principale du programme.

Félicitation, vous venez d'écrire votre premier protocole multi-processus complexe !

TP7 – Fichiers partagés (1/2)

Pour faire les exercices, vous avez besoin de connaître le langage **bash**. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Vous aurez aussi besoin des scripts **`P.sh`** et **`V.sh`**.

Objectifs :

- *comprendre et manipuler des fichiers partagés;*
- *comprendre les notions de sections critiques et de mutex;*
- *savoir utiliser les commandes **`P.sh`** et **`V.sh`**.*

Exercice 1 : Mise en évidence des incohérences provoquées par les commutations (~0h30)

Soit le script **`ecriture.sh`** suivant :

ecriture.sh

```
#! /bin/bash

if [ $# -lt 1 ] ; then
  echo "Il faut au moins un parametre"
  exit 1
fi
for elem in "$@" ; do
  if [ ! -e "$elem" ] ; then
    echo premier $$ > "$elem"
  else
    echo suivant $$ >> "$elem"
  fi
done
```

Question 1.a. Exécutez deux fois de suite la commande **`./ecriture.sh a b c`** et expliquez le contenu des fichiers.

Question 1.b. Soit le script **`lancement_ecriture.sh`** suivant qui permet d'exécuter en concurrence deux processus **`ecriture.sh`**.

lancement_ecriture.sh

```
#! /bin/bash

rm -f f1 f2 f3

./ecriture.sh f1 f2 f3 & ./ecriture.sh f1 f2 f3

wait
```

Exécutez ce script jusqu'à ce que le contenu d'un des fichiers **`f1`**, **`f2`** ou **`f3`** ne contienne qu'une seule ligne au lieu de deux, c'est-à-dire, jusqu'à ce qu'une des écritures soit perdue. Expliquez le résultat obtenu.

Question 1.c. Identifiez la section critique dans **`ecriture.sh`**.

Question 1.d. Modifiez le script **`ecriture.sh`** de façon à assurer une exclusion mutuelle sur la section critique.

Question 1.e. Reprenez le code de la question précédente, mais en faisant attention à ne pas bloquer les processus s'ils n'écrivent pas dans le même fichier.
Remarque : *Il est tout à fait possible que la solution que vous avez proposée à la question précédente assure déjà cette propriété. Dans ce cas, vous avez terminé l'exercice, félicitations !*

Exercice 2 : Accès concurrents à un fichier (~1h)

Soit le script **`ajout.sh`** suivant qui prend au moins deux paramètres. Le premier paramètre correspond à un fichier d'index (liste des fichiers créés), les autres sont les noms des fichiers à créer. Le script crée les fichiers dont le nom est passé en paramètre (s'ils n'existent pas) et met à jour le contenu du fichier d'index. Un message d'erreur est affiché si le nombre de paramètres n'est pas correct ou si le premier paramètre correspond à un nom de répertoire.

ajout.sh

```
#! /bin/bash

if [ $# -lt 2 ] ; then
  echo "Il faut au moins deux parametres"
  exit 1
fi

if [ -d "$1" ] ; then
  echo "Le premier parametre ne doit pas etre un repertoire"
  exit 1
fi

fichier=$1
shift

for i in "$@" ; do
  echo "ajout : test l'existence de $i"
  if [ ! -f "$i" ] ; then
    # dort une seconde pour voir l'exécution suggérée en question b
    sleep 1

    echo "ajout : crée le fichier $i"
    touch "$i"

    # décommentez cette ligne pour voir l'exécution suggérée en question e
    # sleep 2

    echo "ajout : ajoute $i à $fichier"
    echo "$i" >> "$fichier"
   fi
done
```

Question 2.a. Quel sont les contenus du fichier **`index`** et du répertoire après cette exécution :

```
$ ls
ajout.sh
$ ./ajout.sh index fic1 fic2 fic1 fic3
```

Question 2.b. Lancez l'exécution suivante :

```
$ rm -f index fic1 fic2 fic3; ./ajout.sh index fic1 & ./ajout.sh index fic1
```

Observez les contenus du répertoire et du fichier **index** après cette exécution et expliquez pourquoi **fic1** apparaît deux fois dans **index**.

Remarque : *Il est possible, avec une très faible probabilité, que* ***fic1*** *n'apparaisse pas deux fois dans* ***index****. Si par hasard c'était votre cas, relancez simplement l'exécution (et pensez à jouer au loto aujourd'hui car la probabilité est vraiment extrêmement faible).*

Question 2.c. Modifiez le script **ajout.sh** pour que le problème identifié à la question précédente ne se pose plus, c'est-à-dire pour que le fichier **index** contienne exactement la liste des fichiers créés, chacun apparaissant en un unique exemplaire (supprimer l'instruction **sleep** n'est bien sûr pas une solution, ceci ne résout pas le problème de façon générale !).

Question 2.d. On considère maintenant le script **ajout.sh** de l'exercice précédent et le script **enleve.sh** suivant qui prend au moins deux paramètres. Le premier paramètre correspond au fichier d'index et les autres à une liste de fichiers à supprimer. Nous faisons l'hypothèse que le fichier d'index est à jour (tout fichier existant y est référencé). Ce script supprime, s'ils existent, les fichiers dont le nom est passé en paramètre et met à jour le contenu du fichier d'index. Un message d'erreur est affiché si le nombre de paramètres n'est pas correct ou si le fichier d'index n'existe pas.

enleve.sh

```
#! /bin/bash

# ce sleep 2 permet de laisser le temps à ajout.sh de s'exécuter
sleep 2

if [ $# -lt 2 ] ; then
  echo "Il faut au moins deux parametres"
  exit 1
fi

fichier=$1

if [ ! -f "$fichier" ] ; then
  echo "Probleme : le fichier d'index $fichier est introuvable"
  exit 1
fi

shift

for i in "$@" ; do
  echo "enleve : test l'existence de $i"
  if [ -f "$i" ] ; then
    echo "enleve : supprime le fichier $i"
    rm -f "$i"
    echo "enleve : enleve $i de $fichier"
    grep -v "^$i$" "$fichier" > "$fichier".tmp
    mv "$fichier".tmp "$fichier"
  fi
```

```
done
```

Remarque : *Le* **"^i"** *suivant le* **grep -v** *signifie une ligne commençant par* **$i** *(^ initial) et se terminant par* **$i** *($ final).* **$i** *étant l'un des arguments (i.e., un nom de fichier), le motif identifie une ligne contenant uniquement le nom du fichier. Conjointement avec le* **-v**, *la commande* **grep** *renvoie toute les lignes ne contenant pas le fichier passé en paramètre.*

Soit le script **essai1.sh** suivant :

essai1.sh

```
#! /bin/bash

rm -f fic1
rm -f index
touch index

./ajout.sh index fic1
./enleve.sh index fic1
```

Quel sont les contenus du fichier **index** et du répertoire après l'exécution des commandes suivantes ?

```
$ ls
ajout.sh        enleve.sh       essai1.sh
$ ./essai1.sh
```

Question 2.e. Pour cette question on ne considère aucune contrainte sur la localisation des commutations de processus, elles peuvent avoir lieu **après n'importe quelle instruction**.
Soit le script **essai2.sh** suivant :

essai2.sh

```
#! /bin/bash

rm -f fic1
rm -f index
touch index

./ajout.sh index fic1 &
./enleve.sh index fic1 &

wait
```

Est-il possible d'avoir l'exécution suivante ? Vous pouvez la faire apparaître en décommentant le sleep 2 de **ajout.sh**. Expliquez la suite d'événements menant à cette exécution.

```
$ ls
ajout.sh        enleve.sh       essai2.sh       index
$ cat index
$ ./essai2.sh
$ ls
ajout.sh        enleve.sh       essai2.sh       index
```

```
$ cat index
fic1
```

Question 2.f. Modifiez le script **`enleve.sh`** pour que le problème identifié à la question précédente ne se pose plus.

Question 2.g. La solution de la question précédente permet-elle d'assurer que l'exécution du script **`essai2.sh`** donnera toujous le même résultat ? Expliquez votre réponse.

Exercice 3 : Accès concurrents à plusieurs fichiers (~1h)

Soit une application nécessitant l'identification d'utilisateurs et le stockage d'informations les concernant. Les informations sur les utilisateurs sont stockées dans trois fichiers différents :

- le fichier **`login.txt`** contient l'identifiant de connexion de chaque utilisateur connu du système à raison d'un identifiant par ligne,
- le fichier **`pass.txt`** contient le mot de passe de chaque utilisateur, à raison d'un mot de passe par ligne,
- le fichier **`nom.txt`** contient le nom de chaque utilisateur, à raison d'un nom par ligne.

Les informations sont associées en fonction de leur position dans les fichiers (les informations se trouvant à la i<sup>ème/sup> ligne de chaque fichier concernent le même utilisateur).

Le script **`creation_utilisateur.sh`** suivant est une première version du script de création d'un nouvel utilisateur :

creation_utilisateur.sh

```
#! /bin/bash

if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ] ; then
  echo "Vous devez saisir un identifiant, un mot de passe et un nom non vide"
  exit 1
fi

if [ -f login.txt ] && grep "$1" login.txt > /dev/null ; then
  echo "Choisissez un identifiant different de $1"
  exit 1
fi

echo "$1" >> login.txt
echo "$2" >> pass.txt
echo "$3" >> nom.txt
```

Question 3.a. Nous considérons que la base de fichiers est incorrecte si :

- deux utilisateurs ont pu être créés avec le même identifiant (deux lignes identiques dans le fichier **`login.txt`**),
- les informations concernant un utililisateur ne sont pas à la même ligne dans les trois fichiers.

Le premier cas peut se produire lors de l'exécution de :

```
./creation_utilisateur.sh l p u & ./creation_utilisateur.sh l p u
```

et le second lors de l'exécution de :

```
./creation_utilisateur.sh l1 p1 u1 & ./creation_utilisateur.sh l2 p2 u2
```

Donnez, pour chaque cas, un ordonnancement pouvant conduire à une base de fichiers incorrecte.

TP8 – Fichiers partagés (2/2)

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Vous aurez aussi besoin des scripts `P.sh` et `V.sh`.

Objectifs :

- *comprendre et manipuler des fichiers partagés;*
- *comprendre les notions de sections critiques et de mutex;*
- *savoir utiliser les commandes `P.sh` et `V.sh`;*
- *comprendre le problème d'inter-blocage.*

Exercice 1 : Problème de la piscine (~2h30)

Objectifs :

- *s'initier au concept de sémaphore (compteur et file d'attente),*
- *proposer une mise en œuvre approchée.*

Quelques mots de présentation du concept de sémaphore

Dans cet exercice, nous étudions le concept de sémaphore. Comme indiqué par l'auteur du concept, Edsger Dijkstra, dans l'annexe de son article CACM de mai 1968, un sémaphore est une structure de données composée d'un compteur (un entier) et d'une file d'attente (de processus). De manière intuitive :

- lorsque la valeur du compteur est supérieure à **`0`**, un processus peut décrémenter la valeur du compteur du sémaphore et continuer son exécution ;
- lorsque le compteur atteint la valeur **`0`**, un processus peut décrémenter la valeur du compteur du sémaphore mais il devient bloqué et se retrouve dans la file d'attente ;
- lorsque le compteur redevient positif, un processus de la file d'attente est débloqué et retiré de la file d'attente.

Nous émulons le concept de sémaphore avec les scripts **`P.sh`** et **`V.sh`**. Notre mise en œuvre est une émulation qui s'approche du concept dans le sens où notre solution ne sera pas complète. Le concept est mis en œuvre dans le noyau UNIX et peut être utilisé par exemple en langage C : les curieux peuvent exécuter la commande **`apropos semaphore`** et à l'issue de cette séance parcourir les pages du manuel en ligne.

Vue globale de l'application

Nous considérons un ensemble de processus représentant les usagers d'une piscine. La création d'un processus correspond à l'arrivée de l'usager à la piscine. L'usager doit, dans cet ordre :

1. acquérir une cabine disponible ;
2. acquérir un panier pour y déposer ses vêtements ;
3. se changer dans la cabine et la libérer ;
4. nager ;
5. acquérir une cabine disponible ;
6. sortir de son panier ses affaires et libérer le panier ;
7. libérer la cabine.

La figure 1 présente graphiquement l'ensemble des scripts et fichiers que vous allez manipuler.

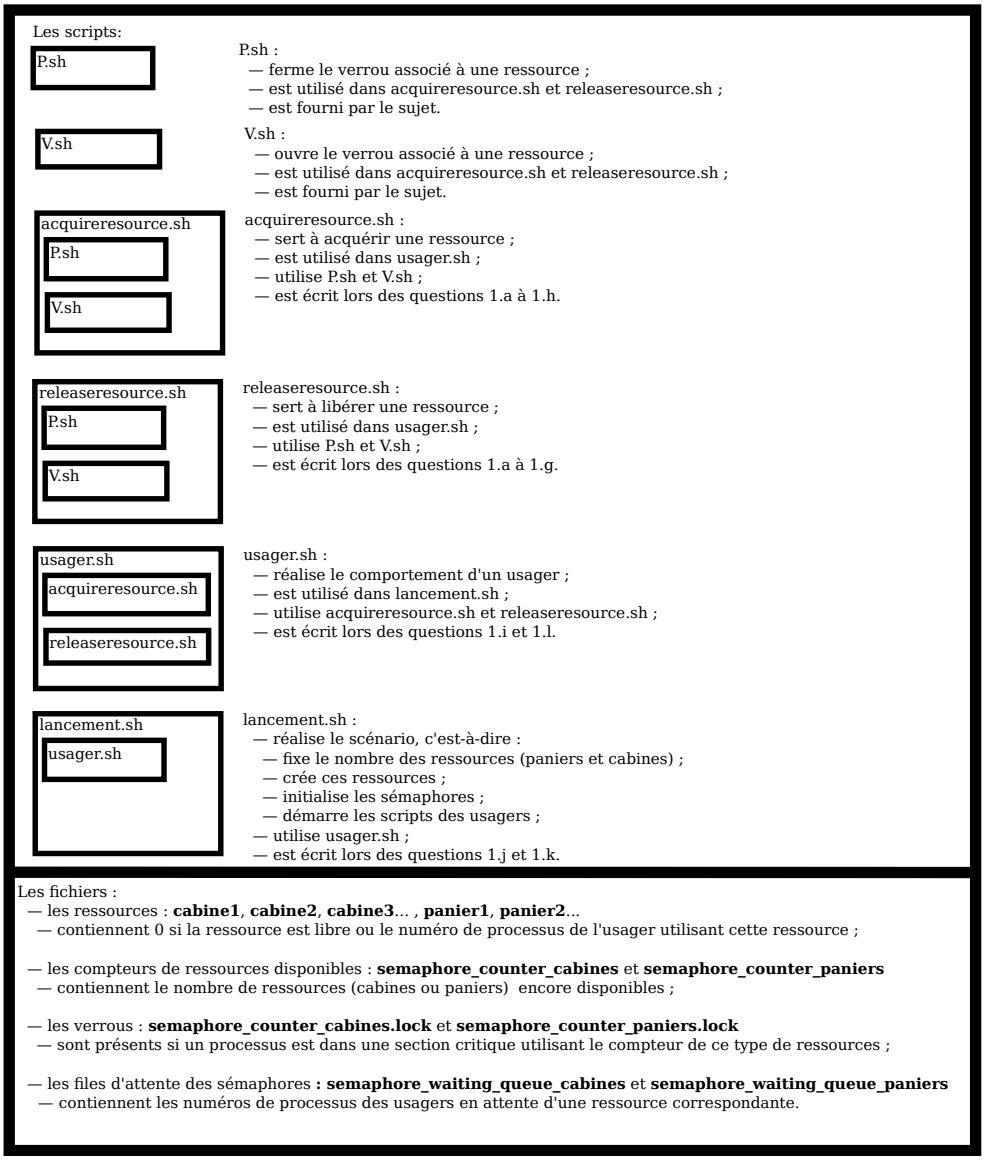


Figure 1 – Scripts et fichiers nécessaires à la réalisation de l'exercice.

Partie 1 : Gestion des ressources

Pour suivre la disponibilité des ressources (cabines et paniers), nous disposons d'un fichier par type de ressource (**`semaphore_counter_cabines`** pour les cabines et **`semaphore_counter_paniers`** pour les paniers). Chacun des fichiers contient un entier qui correspond au nombre de ressources disponibles, c'est-à-dire au compteur du sémaphore du type de ressource. Par ailleurs, nous ajoutons un fichier par type de ressource pour mémoriser les processsus en attente d'une ressource de ce type (**`semaphore_waiting_queue_cabines`** pour les cabines et **`semaphore_waiting_queue_paniers`** pour les paniers). Ce sont les files d'attente des sémaphores. Enfin, à chaque ressource correspond un fichier (**`cabine1`**, **`cabine2`**, etc., et **`panier1`**, **`panier2`**, etc.). Ces fichiers contiennent un entier : soit **`0`** lorsque la ressource est disponible, soit le numéro du processus qui détient la ressource.

Question 1.a. Gestion du compteur du sémaphore sans section critique Pour commencer, nous ignorons la gestion des files d'attente des sémaphores ainsi que la mémorisation de quel processus possède quelles ressources.

Notez que nous forgeons le nom du fichier contenant la valeur compteur du sémaphore avec le nom du type de la ressource. Ce nom sera passé comme premier paramètre des scripts **`acquireresource.sh`** et **`releaseresource.sh`**. Ainsi, utilisez l'expression suivante pour forger le nom du fichier : **`"semaphore_counter_${1}s"`**, sachant que **`${1}`** est le nom du type de la ressource demandée.

Remarque : *L'utilisation de la forme* ***`${nom_variable}`*** *n'a pas été vue en cours, mais n'est pas très difficile à comprendre. Elle permet ici de séparer le nom de la variable du texte qui suit.*

Pour acquérir une ressource du type *RT* (avec *RT* égal **cabine** ou **panier**), il faut :

1. lire le contenu du fichier du compteur du sémaphore (motif **semaphore_counter_*RT*s**) pour savoir combien de ressources sont disponibles ;
2. tant que la valeur lue n'est pas au moins égale à 1, forcer une commutation avec la commande **sleep 1** en espérant la libération d'une ressource pendant cette période, puis lire à nouveau le contenu du fichier ;
3. mettre à jour le contenu du fichier **semaphore_counter_*RT*s** pour signaler qu'une ressource de moins est disponible.

Pour libérer une ressource du type *RT*, il faut :

1. ajouter un à la valeur contenue dans le contenu du fichier du compteur du sémaphore (motif **semaphore_counter_*RT*s**) pour signaler qu'une ressource de plus du type *RT* est disponible.

Écrivez les script **acquireresource.sh** et **releaseresource.sh** qui permettent d'acquérir et de libérer une ressource. Ces scripts prennent en paramètre le type la ressource demandée. Dans le cas de la piscine, par exemple, les paramètres seront **cabine** et **panier**. Pensez à tester votre code sur des fichiers factices que vous aurez initialisés. Par exemple avec le scénario suivant :

```
$ echo 1 > semaphore_counter_cabines
$ ./acquireresource.sh cabine
$ cat semaphore_counter_cabines
0
$ ./acquireresource.sh cabine
    # bloque tant que vous n'avez pas invoqué ./releaseresource.sh cabine dans un autre terminal
```

Question 1.b. Que peut-il se produire si deux processus invoquent **acquireresource.sh** ou **releaseresource.sh** en parallèle ? Donnez un scénario menant à une incohérence.

Question 1.c. Identifiez les sections critiques dans les scripts **acquireresource.sh** et **releaseresource.sh** sans tenter de corriger le problème de synchronisation (la correction du problème est l'objet des questions suivantes).

Question 1.d. Gestion du compteur du sémaphore avec section critique Comme première solution, erronée, nous vous proposons d'acquérir un verrou au début de chaque section critique et de le relâcher à la fin. De façon à assurer un maximum de parallélisme, vous veillerez à ce que deux processus accédant à des sémaphores différents n'utilisent pas le même verrou. Vous pouvez, par exemple, forger le nom du verrou avec **"semaphore_counter_${1}s.lock"**, sachant que **${1}** est le nom du type de la ressource demandée.

Pour tester vos scripts, initialisez votre compteur du sémaphore à une grande valeur, par exemple **100**.

Question 1.e. On vous propose d'exécuter le scénario suivant, en commençant par lancer toutes les commandes du terminal 1 avant de lancer celles du terminal 2 :

Premier terminal	Second terminal

`$ echo 1 > semaphore_counter_cabines`
`$ ./acquireresource.sh cabine`
`$ cat semaphore_counter_cabines`
`0`
`$ ./acquireresource.sh cabine # doit bloquer`
	`$ ./releaseresource.sh cabine`
	`  # doit aussi bloquer`

Commencez par vérifier que les dernières commandes de chaque terminal sont bien bloquées. Ensuite, expliquez pour quelle raison le **./releaseresource.sh** lancé dans le terminal 2 est bloqué, et est donc incapable de libérer une ressource

pour débloquer le **./acquireresource.sh** du terminal 1.

Question 1.f. Modifiez le script **acquireresource.sh** de façon à vous assurez que le scénario décrit à la question précédente ne conduise plus à un interblocage.

*Attention : Pour quitter votre programme à la question précédente, vous avez probablement du interrompre brutalement vos scripts qui n'ont donc pas pu libérer les verrous. Ces verrous sont donc toujours occupés au début de cette question, c'est pourquoi, avant d'effectuer le moindre test, nous vous conseillons de supprimer tous les verrous avec la commande **rm -f *.lock** et d'exécuter une commande **ps** pour regarder s'il ne reste pas des processus en suspens.*

Question 1.g. Gestion des occupations des ressources Nous ajoutons maintenant autant de fichiers qu'il y a de ressources et écrivons dans les fichiers des ressources les numéros de processus qui possèdent les ressources, avec le cas particulier **0** lorsqu'une ressource est disponible.

*Remarque : un appel au script **acquireresource.sh** n'acquiert qu'une seule ressource. Il en va de même pour **releaseresource.sh**.*

Dans le script **acquireresource.sh**, avant de mettre à jour le compteur du sémaphore, ajoutez ce qui suit :

1. parcourez l'ensemble des fichiers des ressources du type *RT* passé en argument (motif ***RT**** avec *RT* égal à **cabine** ou **panier**) pour trouver une ressource libre (un fichier avec la valeur **0**) ;
2. mettez à jour le fichier trouvé avec l'identité du processus demandant la ressource, c'est-à-dire avec l'identité du processus appelant le script **acquireresource.sh** (repéré avec la variable PPID [on utilise **PPID** et non **$$** puisque c'est le père du processus **acquireresource.sh** qui doit acquérir la ressource) ;
3. lorsque le fichier a été trouvé et mis à jour, sortez de la boucle avec l'instruction **break** afin de n'acquérir qu'une ressource.

*Remarque : dans les questions à venir, le script **acquireresource.sh** sera placé dans un script représentant les actions d'un usager (acquérir une cabine, etc.). Ainsi, le **PPID** identifiera l'usager. En conséquence, il ne faut pas s'inquiéter lorsque, à ce niveau du sujet, nous retrouvons N fois le même PPID propriétaire de plusieurs ressources dans les scénarios d'exécution.*

Dans le script **releaseresource.sh**, avant de mettre à jour le compteur du sémaphore, ajoutez ce qui suit :

1. parcourez l'ensemble des fichiers des ressources du type *RT* passé en argument (motif ***RT**** avec *RT* égal à **cabine** ou **panier**) pour trouver la ressource prise par le processus appelant (le fichier avec la valeur de **PPID**) ;
2. mettez à jour le fichier trouvé avec **0** pour indiquer que la ressource est de nouveau disponible ;
3. lorsque le fichier a été trouvé et mis à jour, sortez de la boucle avec l'instruction **break** afin de ne rendre qu'une ressource.

*Attention : dans le scénario qui suit, il faut que ce soit le même shell qui exécute les scripts **acquireresource.sh** et **releaseresource.sh**, ceci afin d'avoir le même **PPID**. Donc, contrairement aux premières exécutions en début d'exercice, les commandes sont à exécuter dans le même terminal, avec certaines commandes en arrière plan.*

Pensez à tester votre code sur des fichiers factices que vous aurez initialisés. Par exemple selon le scénario suivant :

```
$ echo 2 > semaphore_counter_cabines
$ echo 0 > cabine1; echo 0 > cabine2
$ echo $$
10886
$ ./acquireresource.sh cabine
$ cat cabine1 cabine2
10886
0
$ ./acquireresource.sh cabine
```

```
$ cat semaphore_counter_cabines
0
$ cat cabine1 cabine2
10886
10886
# attention, le prochain en arrière plan dans le même terminal
$ ./acquireresource.sh cabine &
$ jobs
[2]+  Stoppé                 ./acquireresource.sh cabine
$ ./releaseresource.sh cabine
$ ./releaseresource.sh cabine
$ cat semaphore_counter_cabines
1
$ cat cabine1 cabine2
0
10886
$ ./releaseresource.sh cabine
$ cat semaphore_counter_cabines
2
$ cat cabine1 cabine2
0
0
```

Question 1.h. Gestion des files d'attente des sémaphores Nous terminons avec l'ajout des files d'attente. Un processus qui est bloqué en attente d'une ressource du type *RT* est présent dans la file d'attente du sémaphore correspondant. Nous proposons de forger le nom du fichier de la file d'attente du sémaphore avec **`"semaphore_waiting_queue_${1}s"`**.

Modifiez le script **`acquireresource.sh`** comme suit :

1. ajoutez le numéro du processus appelant le script (**`PPID`**) dans le fichier **`semaphore_waiting_queue_${1}s`** juste avant la boucle d'attente d'ouverture du verrou **`semaphore_counter_${1}s`** ;
2. retirez le numéro du processus juste après la même boucle, c'est-à-dire lorsque le processus est garanti d'avoir une ressource. Pour le retrait, vous utilisez la même procédure que celle proposée dans le TP précédent : **`grep -v "^$PPID$" origine > origine.tmp`** puis **`mv origine.tmp origine`**.

Attention : *dans le scénario qui suit, il faut que ce soit le même shell qui exécute les scripts* ***`acquireresource.sh`*** *et* ***`releaseresource.sh`****, ceci afin d'avoir le même* ***`PPID`****. Donc, contrairement aux premières exécutions en début d'exercice, les commandes sont à exécuter dans le même terminal, avec certaines commandes en arrière plan.*

Pensez à tester votre code sur des fichiers factices que vous aurez initialisés. Par exemple selon le scénario suivant :

```
$ echo 2 > semaphore_counter_cabines
$ echo 0 > cabine1; echo 0 > cabine2
$ echo $$
13049
$ ./acquireresource.sh cabine
$ cat cabine1 cabine2
13049
0
$ cat semaphore_counter_cabines
1
$ cat semaphore_waiting_queue_cabines
```

```
$ ./acquireresource.sh cabine
$ cat semaphore_counter_cabines
0
$ cat cabine1 cabine2
13049
13049
# attention, le prochain en arrière plan dans le même terminal
$ ./acquireresource.sh cabine &
[2] 10657
$ jobs
[2]+  Stoppé                 ./acquireresource.sh cabine
$ cat semaphore_waiting_queue_cabines
13049
$ ./releaseresource.sh cabine
$ cat semaphore_waiting_queue_cabines
[2]+  Fini                    ./acquireresource.sh cabine
$ cat semaphore_counter_cabines
0
$ cat cabine1 cabine2
13049
13049
$ ./releaseresource.sh cabine
$ cat semaphore_counter_cabines
1
$ cat cabine1 cabine2
0
13049
$ cat semaphore_waiting_queue_cabines
$ ./releaseresource.sh cabine
$ cat semaphore_counter_cabines
2
$ cat cabine1 cabine2
0
0
$ cat semaphore_waiting_queue_cabines
```

Bravo, vous venez de mettre en œuvre votre première primitive de synchronisation : un sémaphore avec son compteur et sa file d'attente !
Nous pouvons maintenant faire des expériences d'interblocage.

Partie 2 : Gestion de la piscine

Question 1.i. En utilisant vos scripts **`acquireresource.sh`** et **`releaseresource.sh`**, mettez en œuvre l'algorithme de l'usager de la piscine dans un script **`usager.sh`**. Symbolisez le fait de se baigner en affichant **`PID se baigne`**, où **`PID`** est le PID du processus **`usager.sh`**, et en exécutant la commande **`sleep 5`** qui endort le processus pendant **5** secondes.
Voici pour rappel l'algorithme de l'usager proposé en début d'énoncé :

1. acquérir une cabine disponible ;
2. acquérir un panier pour y déposer ses vêtements ;
3. se changer dans la cabine et la libérer ;

4. nager ;
5. acquérir une cabine disponible ;
6. sortir de son panier ses affaires et libérer le panier ;
7. libérer la cabine.

Question 1.j. Écrivez un programme **`lancement.sh`** qui :

- initialise **`semaphore_counter_paniers`** à **5** ;
- initialise **`semaphore_counter_cabines`** à **3** ;
- crée les fichiers des files d'attente (**`semaphore_waiting_queue_cabines`** et **`semaphore_waiting_queue_paniers`**) ;
- crée les paniers et les cabines (n'oubliez pas de mettre **0** comme valeur initiale dans tous ces fichiers) ;
- démarre **7** usagers en parallèle ;
- attend la fin de tous les processus usagers ; et
- affiche **`Processus lancement se termine`** à la fin de votre programme.

Lancez **`lancement.sh`**. Vérifiez que l'affichage est cohérent. Vérifiez aussi que, après la terminaison de **`lancement.sh`**, **`semaphore_counter_paniers`** contient bien **5** et **`semaphore_counter_cabines`** **3**.

Afin de faciliter la conception, nous proposons le script **`cleanup.sh`** à exécuter entre deux exécutions du script **`lancement.sh`**.

Question 1.k. Modifiez le script **`lancement.sh`** de façon à lancer **10** usagers au lieu de **7**. Qu'observez-vous ? Donnez une explication.
Attention : cette question est la question clé de l'exercice ! Prenez le temps d'argumenter votre réponse par écrit. Et en séance, n'hésitez pas à la valider avec un encadrant.

Question 1.l. Que se passe-t-il, si on inverse l'ordre de prise du panier et de la cabine, c'est-à-dire, un usager commence par acquérir un panier avant d'acquérir une cabine pour se changer, libère alors la cabine, se baigne, prend à nouveau une cabine, se change, puis libère le panier et la cabine. Modifiez le script **`usager.sh`** pour vérifier expérimentalement votre hypothèse.

Question 1.m. Détection d'interblocage — défi Écrivez le script **`detectdeadlock.sh`** qui détecte s'il y a un interblocage pendant une exécution.

Remarque : *nous pouvons écrire un tel script car la propriété d'interblocage est dite stable (non fugace), c'est-à-dire que, lorsqu'il y a un interblocage, le système reste en interblocage tant que l'on n'intervient pas pour résoudre le problème (par exemple en supprimant des processus ou en ajoutant des ressources).*

Complétez le script **`lancement.sh`** de façon à vérifier régulièrement que la piscine n'est pas dans un état d'interblocage. Vous veillerez (1) à ce que le script **`lancement.sh`** se termine lorsque tous les usagers sont sortis de la piscine et (2) à ce que le script qui détecte l'interblocage arrête le script **`lancement.sh`**.

Voici quelques éléments d'aide pour la conception de cette dernière version :

- pour arrêter les processus de détection ou de lancement du scénario, pensez à utiliser les signaux : lorsque l'algorithme de détection établit l'interblocage, le script **`detectdeadlock.sh`** envoie un signal **`USR1`** au script **`lancement.sh`**, et inversement lorsque le script **`lancement.sh`** arrive à la fin ;
- le **`wait`** à la fin du script **`lancement.sh`** doit être modifié pour attendre uniquement les processus **`usager.sh`** ;
- dans le script de détection, pensez à acquérir les verrous sur les deux compteurs afin d'utiliser un état cohérent dans l'algorithme de détection ;
- un processus est « inter-bloqué » lorsque'il est en attente d'une ressource du type *TR1* alors qu'il possède une ressource du type *TR2*. Un interblocage est détecté lorsque tous les processus possédant une ressource sont « inter-bloqués », c'est-à-dire, dans notre étude de cas, en attente d'une ressource de l'autre type. Autrement dit, il y a un

interblocage du sytème lorsque le nombre de processus « inter-bloqués » atteint le nombre de ressources du système, soit le nombre de cabines + le nombre de paniers.

Bravo, vous avez détecté à la main, voire automatiquement, un interblocage dans le modèle le plus complexe, le modèle ET-OU :
« il faut un panier parmi un ensemble et une cabine parmi un ensemble »,
autrement dit « (panier1 ∨ panier2 ∨ panier3 ∨ panier4 ∨ panier5) ∧ (cabine1 ∨ cabine2 ∨ cabine3) » !

Exercice 2 : Problème du pont (~1h – optionnel)

Imaginez un pont routier à une seule voie. Des voitures se présentent aux deux extrémités du pont, notre but est d'écrire les contrôleurs présents à chaque extrémité du pont. Ces contrôleurs doivent assurer que deux voitures circulant dans des directions opposées ne se trouvent pas en même temps sur le pont. Nous allons bien sûr nous intéresser à une version simplifiée des contrôleurs.

Voici une proposition de scripts pour les contrôleurs. Le script **`EstOuest.sh`** représente le contrôleur qui gère les accès au pont pour une traversée d'est en ouest, le script **`OuestEst.sh`** gère les accès au pont pour une traversée d'ouest en est.

Remarque : *La commande* ***`touch fic`*** *que vous ne connaissez pas encore s'occupe simplement de créer un fichier* ***`fic`*** *vide.*

Remarque : *Nous vous rappelons que la boucle* ***`while [ ! -f EstOuest.fic ]; ...; done`*** *continue tant que le fichier* ***`EstOuest.fic`*** *n'existe pas puisque* ***`-f EstOuest.sh`*** *renvoie vrai si le fichier existe et* ***`! -f EstOuest.fic`*** *inverse la proposition. Cette boucle permet donc d'attendre que le fichier* ***`EstOuest.fic`*** *soit créé.*

EstOuest.sh

```
#! /bin/bash
# EstOuest.sh

i=0

while [ $i -lt $1 ]; do
  while [ ! -f EstOuest.fic ]; do
    sleep 1
  done

  i=$(expr $i + 1)
  echo "Route Est-Ouest ouverte, $i voitures sur $1 sont passées"

  rm -f EstOuest.fic
  touch OuestEst.fic
done
```

OuestEst.sh

```
#! /bin/bash
# OuestEst.sh

i=0

while [ $i -lt $1 ]; do
  while [ ! -f OuestEst.fic ]; do
    sleep 1
  done

  i=$(expr $i + 1)
  echo "Route Ouest-Est ouverte, $i voitures sur $1 sont passées"

  rm -f OuestEst.fic
  touch EstOuest.fic
done
```

Question 2.a. Expliquez ce qui se passe lors de l'exécution du script suivant :

lancement.sh

```
#! /bin/bash
# lancement.sh

killall EstOuest.sh OuestEst.sh
```

```
if [ -f EstOuest.fic ]
then
  rm EstOuest.fic
fi
touch OuestEst.fic

./OuestEst.sh 3 &
./EstOuest.sh 3 &

wait
```

Question 2.b. On modifie maintenant le script **lancement.sh** pour faire passer trois voitures d'ouest en est et quatre voitures d'est en ouest :

lancement.b.sh

```
#! /bin/bash
# lancement.sh

killall EstOuest.sh OuestEst.sh

if [ -f EstOuest.fic ]
then
  rm EstOuest.fic
fi
touch OuestEst.fic

./OuestEst.sh 3 &
./EstOuest.sh 4 &

wait
```

Expliquez pourquoi le programme se bloque.

Question 2.c. De façon à éviter le blocage identifié à la question précédente, on vous propose d'utiliser un fichier nommé **pas-fini** pour indiquer qu'aucun des deux scripts **EstOuest.sh** et **OuestEst.sh** ne sont terminés. Pour mettre en œuvre cet algorithme, vous devez modifier vos scripts de la façon suivante :

- avant de créer **EstOuest.sh** et **OuestEst.sh**, **lancement.sh** doit créer le fichier **pas-fini** pour indiquer qu'aucun des deux scripts n'est terminé;
- avant de faire passer une voiture, **EstOuest.sh** (resp. **OuestEst.sh**) attend soit que le fichier **EstOuest.fic** (resp. **OuestEst.fic**) soit présent, soit que le fichier **pas-fini** ne soit pas présent;
- **EstOuest.sh** et **OuestEst.sh** détruisent le fichier **pas-fini** avant de se terminer.

Mettez en œuvre ce nouvel algorithme.

Exercice 3 : Producteur/consommateur (~ 1h – défi)

Question 3.a. Écrivez un script **ecrivain.sh** qui prend au moins 2 paramètres. Le premier paramètre est une chaîne de caractères. Si la valeur de ce paramètre correspond à un répertoire existant, le script se termine avec un message d'erreur. Sinon, le script écrit successivement les valeurs des autres paramètres dans un fichier dont le nom correspond au premier paramètre. La suite de commandes :

```
$ ls
ecrivain.sh     lecteur.sh
$ ./ecrivain.sh fic_test 3 2 6 4
```

a donc pour effet de créer un fichier **fic_test** qui contient les valeurs 3, 2, 6, 4 (une par ligne) à la fin de l'exécution. Si le fichier existait déjà, son contenu est remplacé.

Question 3.b. Écrivez un script **lecteur.sh** qui prend exactement 1 paramètre. Si ce paramètre ne correspond pas à un fichier, le script termine son exécution avec un message d'erreur. Sinon, il lit le contenu du fichier ligne par ligne. Pour chaque ligne lu, il affiche **ligne lue :** suivi de la valeur lue.

Pour pouvoir lire un fichier ligne à ligne, vous utiliserez la construction suivante :

```
( while read x; do ... done ) < fichier
```

Pour comprendre cette construction, vous devez savoir que :

- la parenthèse permettant de regrouper les instructions, la redirection du flux d'entrée à partir du fichier **fichier** s'adresse à l'ensemble des instructions entre parenthèse, soit l'ensemble du code **while ... done** (notez que techniquement, le **while ... done** est vu comme une unique commande par **bash** et que les parenthèses sont donc optionnelles);
- à chaque tour de boucle, **read** lit une ligne à partir du fichier (puisque le fichier sert de flux d'entrée), et avance d'autant la tête de lecture du flux, comme si on avait écrit les lignes les unes à la suite des autres dans le terminal;
- lorsque la tête de lecture n'a pas atteint la fin du fichier, la commande **read** renvoie vrai (i.e., la valeur 0) alors que lorsque la tête de lecture atteint la fin du fichier, la commande **read** renvoie faux (i.e., une valeur différente de 0), ce qui arrête la boucle;

Question 3.c. Nous souhaitons maintenant synchroniser le lecteur et l'écrivain de manière à ce que chaque valeur écrite dans le fichier soit ensuite lue, mais sans imposer une exécution séquentielle du lecteur et de l'écrivain. Pour mettre en évidence les problèmes qui peuvent se produire, nous ajoutons une instruction **sleep 1** dans la boucle d'écriture. Exécutez plusieurs fois la commande suivante (en n'oubliant pas de détruire le fichier **fic_test** entre deux exécutions). Que constatez-vous ?

```
$ ./ecrivain.sh fic_test 1 2 3 4 5 & ./lecteur.sh fic_test &
```

Question 3.d. Nous souhaitons maintenant que le lecteur lise (et donc affiche) chacune des valeurs écrites par l'écrivain. Pour obtenir ce résultat, nous allons forcer une alternance stricte entre les écritures et les lectures. Il faut donc garantir que :

- la première opération effectuée soit une écriture;
- l'écrivain attende pour faire une nouvelle écriture que son écriture précédente ait été lue;
- le lecteur ne fasse pas de nouvelle lecture s'il n'y a pas eu de nouvelle écriture.

Lorsqu'il a terminé ses écritures, l'écrivain écrit la chaîne **fin** dans le fichier. Lorsqu'il lit cette valeur, le lecteur sait donc qu'il peut se terminer.

Techniquement, pour synchroniser l'écrivain et le lecteur, on vous demande d'utiliser des fichiers nommés **"$1-ecr.sync"** et **"$1-lec.sync"**, où **$1** correspond donc au fichier dans lequel les données sont écrites par l'écrivain. L'écrivain doit attendre que le fichier **"$1-ecr.sync"** existe avant d'effectuer une écriture, et le lecteur doit attendre que le fichier **"$1-lec.sync"** existe avant d'effectuer une lecture. Après son écriture (resp. sa lecture), l'écrivain supprime le fichier **"$1-ecr.sync"** (resp. **"$1-lec.sync"**) et crée le fichier **"$1-lec.sync"** (resp. **"$1-ecr.sync"**) pour débloquer le lecteur (resp. l'écrivain). Notez que comme la première opération a effectuer est une écriture, c'est au lecture de créer le premier fichier **"$1-ecr.sync"**.

Modifiez les deux scripts pour qu'ils se synchronisent correctement.

TP9 – Révision

Pour faire les exercices, vous avez besoin de connaître le langage **bash**. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

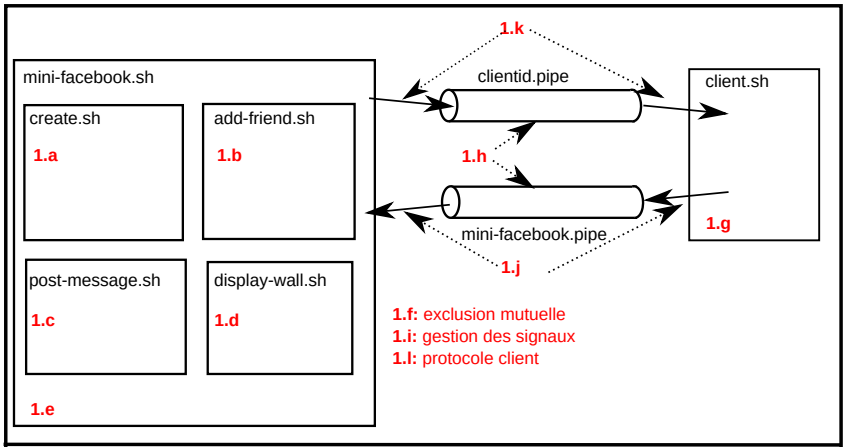
Vous aurez aussi besoin des scripts **`P.sh`** et **`V.sh`**.

Objectifs :

- *revoir la plupart des notions étudiées dans le module.*

Exercice 1 : Mini Facebook (~ 3h)

Le but de cet exercice est de concevoir un mini serveur facebook. La figure 2 vous présente l'architecture globale de l'application, n'hésitez pas à vous y référer tout au long de l'exercice pour comprendre à quelle étape vous êtes. Les différentes parties vous seront bien sûr expliquées au fur et à mesure. Pour démarrer, il faut comprendre que le script **`mini-facebook.sh`** met en œuvre le serveur et qu'il utilise les scripts **`create.sh`**, **`add-friend.sh`**, **`post-message.sh`** et **`display-wall.sh`**. Le client est mis en œuvre par le script **`client.sh`**, et il communique avec le serveur via deux tubes nommés : **`mini-facebook.sh`** (partagé par tous les clients) et **`clientid.pipe`** (un tube par client).



Les scripts:
- P.sh et V.sh: gestion des verrous (fournis par le sujet) ;
- create.sh: création d'un utilisateur, utilisé par mini-facebook.sh ;
- add-friend.sh: ajout d'un ami, utilisé par mini-facebook.sh ;
- post-message.sh: ajout d'un message sur le mur d'un ami, utilisé par mini-facebook.sh ;
- display-wall.sh: affiche le mur d'un utilisateur, utilisé par mini-facebook.sh ;
- mini-facebook.sh: serveur du service, utilise create.sh, addfriend.sh, post-message.sh et display-wall.sh ;
- client.sh: client du service.

Les fichiers:
- les repertoires **id**: un répertoire par utilisateur (id) crée , contient la liste d'amis (**friends**) et le mur (**wall**) ;
- **id/friends**: listes des amis de id. Chaque ligne contient l'identificateur d'un ami ;
- **id/wall**: listes des messages postés sur le mur de id. Chaque ligne contient un message sous la forme "expéditeur: message" ;
- le tube **mini-facebook.pipe**: tube où tous les clients écrivent les messages destinés au serveur mini-facebook.sh ;
- les tubes **clientid.pipe**: tubes où le serveur mini-facebook.sh écrit tous les messages destinés aux clients (le tube **id.pipe** est lu par le client id).

Figure 2 – Architecture de l'application.

Partie 1 : Mise en place des commandes

Dans un premier temps, nous allons créer l'ensemble des commandes permettant de manipuler l'état du serveur.

Le serveur gère un ensemble d'utilisateurs. Un utilisateur d'identifiant **`$id`** possède un répertoire **`$id`** contenant deux fichiers :

- un fichier nommé **`wall`**, contenant une suite de lignes. Chaque ligne est un message posté par un ami sous la forme **`identifiant-ami: message`** ;
- un fichier nommé **`friends`**, dans lequel chaque ligne contient l'identifiant d'un ami. Seul les amis d'un utilisateur peuvent poster des messages sur son mur.

Le serveur doit être capable d'effectuer quatre actions différentes :

- créer un utilisateur, c'est-à-dire créer le répertoire associé à un utilisateur et initialiser les fichiers de l'utilisateur,
- ajouter un ami à un utilisateur,
- poster un message sur le mur d'un ami,
- lire son mur.

Question 1.a. Vous commencez par écrire un script **`create.sh id`** prenant un unique argument **`$id`** et permettant de créer le répertoire et, dans cet ordre, les fichiers **`wall`** puis **`friends`** de l'utilisateur d'identifiant **`$id`**. En effectuant les actions dans cet ordre, vous pouvez noter que :

- si le répertoire **`$id`** existe, c'est que l'utilisateur est en cours de création,
- si le fichier **`$id/friends`** existe, c'est que l'utilisateur est entièrement créé.

Votre script doit afficher sur le terminal :

- **`nok please provide an identifier`** si le script ne reçoit pas d'argument,
- **`nok user '$id' already exists`** si l'utilisateur est déjà en cours de création, c'est-à-dire si le répertoire **`$id`** existe déjà,
- **`ok`** sinon (n'oubliez pas d'afficher **`ok`**, cet affichage deviendra essentiel par la suite).

Attention : Seuls ces trois affichages sont autorisés. Notez que chacun de ces affichages n'est constitué que d'une unique ligne. Tout autre affichage finira par engendrer des erreurs lorsque vous répondrez aux questions de la fin du TP.

Question 1.b. On souhaite maintenant pouvoir ajouter un ami à la liste des amis d'un utilisateur d'identifiant **`$id`**.

Votre script doit d'abord vérifier que l'utilisateur **`$id`** existe, c'est-à-dire que le répertoire **`$id`** existe.

Ensuite, un ami, identifié par **`$friend`**, est aussi un utilisateur. Avant d'ajouter l'utilisateur **`$friend`** à la liste d'amis de l'utilisateur **`$id`**, il faut donc vérifier que l'utilisateur **`$friend`** existe, c'est-à-dire que le répertoire **`$friend`** existe.

Enfin, avant d'ajouter l'utilisateur **`$friend`** à la liste d'amis de **`$id`**, il faut vérifier que l'utilisateur **`$friend`** n'est pas déjà ami de l'utilisateur **`$id`**, c'est-à-dire que la chaîne de caractère **`$friend`** n'est pas présente dans le fichier d'amis **`$id/friends`** de **`$id`**. Vous pourrez utiliser le code du retour de la commande **`grep`** pour détecter la présence de la chaîne de caractères **`friend`** dans **`$id/friends`**. Vous utiliserez par exemple **`if grep "^$friend$" "$id"/friends > /dev/null; then ... fi`**. La redirection vers **`/dev/null`** est importante pour éviter de polluer la sortie de **`add-friend.sh`** avec la sortie de **`grep`**, ce qui deviendra essentiel par la suite.

Écrivez le script **`add-friend.sh id friend`** prenant les deux arguments **`$id`** et **`$friend`** et ajoutant l'utilisateur **`$friend`** à la liste d'amis de **`$id`**. Votre script doit afficher sur le terminal :

- **`nok user '$id' does not exist`** si l'utilisateur **`$id`** n'a pas été entièrement créé, c'est-à-dire si le fichier **`$id/friends`** n'existe pas.
- **`nok user '$friend' does not exist`** si l'utilisateur **`$friend`** n'a pas été entièrement créé, c'est-à-dire si le fichier **`$friend/friends`** n'existe pas.
- **`nok user '$friend' is already a friend of '$id'`** si l'utilisateur **`$friend`** est déjà un ami de **`$id`**.
- **`ok`** sinon (n'oubliez pas cet affichage).

*Attention : De la même façon, seuls ces affichages (sur une unique ligne) sont autorisés et il faut que **`add-friend.sh`** ne puisse engendrer aucun autre affichage (en particulier, pensez à rediriger la sortie de la commande **`grep`** vers **`/dev/null`**).*

Question 1.c. On souhaite maintenant pouvoir poster des messages. Écrivez un script **`post-message.sh sender receiver ...`** prenant au moins trois arguments. Les deux premiers arguments sont des utilisateurs : **`$sender`** est l'émetteur du message et **`$receiver`** est le récepteur du message. Les arguments suivant constituent le message. Ajouter un message revient à ajouter la ligne **`$sender: ...message...`** dans le mur de l'utilisateur **`$receiver`**, c'est-à-dire dans le fichier **`$receiver/wall`**.

Votre script doit afficher sur le terminal :

- **`nok user '$sender' does not exist`** si l'utilisateur **`$sender`** n'a pas été entièrement créé (fichier **`$sender/friends`** n'existe pas),
- **`nok user '$receiver' does not exist`** si l'utilisateur **`$receiver`** n'a pas été entièrement crée,
- **`nok user '$sender' is not a friend of '$receiver'`** si l'utilisateur **`$sender`** n'est pas un ami de l'utilisateur **`$receiver`** ;
- **`ok`** sinon.

Attention : Comme précédemment, veillez à ne produire que les affichages (d'une unique ligne) décrits ici.

*Remarque : Pensez à utiliser la construction donnée avec **`grep`** à la question précédente.*

Question 1.d. On souhaite enfin pouvoir afficher le mur d'un utilisateur. Écrivez une commande **`display-wall.sh id`** prenant l'identifiant **`$id`** en argument. Votre script doit afficher sur le terminal :

- **`nok user '$id' does not exist`** si **`$id`** n'a pas été entièrement créé ;
- la ligne **`start-of-file`** suivie du contenu du mur de **`$id`** suivi de **`end-of-file`**. Pensez à utiliser la commande **`cat`** pour afficher le contenu du mur.

Remarque : Comme pour les questions précédentes, l'affichage généré par votre programme doit exactement respecter les consignes. Cette commande est la seule qui peut générer des affichages de plus d'une ligne, par exemple :

```
start-of-file
darth-vader: bonjour, tu veux être mon ami ?
C3P0: joli photo
yoda: grosse soirée thème Teletubbies chez Palatine jeudi soir, venez nombreux !
chewbacca: uuuarrrgg !
end-of-file
```

Partie 2 : Mise en place du serveur

On souhaite maintenant mettre en place le serveur. Dans un premier temps, votre serveur interprète des requêtes de façon interactive, c'est-à-dire qu'il lit des requêtes à partir du terminal et les exécute.

Question 1.e. Écrivez un script **`mini-facebook.sh`**. Ce script exécute une boucle infinie. À chaque pas de la boucle, le script lit une requête à partir du terminal. Une requête est toujours sous la forme **`req id args`**.

*Remarque : Pour lire la requête de l'utilisateur, il vous suffit d'utiliser la commande **`read req id args`**. En effet, si l'utilisateur rentre plus de deux mots, **`req`** contiendra le premier mot, **`id`** le second mot, et **`args`** tous les mots qui suivent.*

Votre serveur doit ensuite être capable d'interpréter quatre requêtes différentes en utilisant les quatre scripts que vous avez écrits dans la première partie :

- **`create id`** : crée l'utilisateur **`id`** ;
- **`add id friend`** : ajoute l'ami **`friend`** aux amis de **`id`** ;
- **`post sender receiver message`** : poste un message ;
- **`display id`** : affiche le mur de **`id`**.

Si l'utilisateur écrit une requête incompréhensible, votre serveur affiche **`nok bad request`** dans le terminal.

Pour sélectionner l'action adéquate en fonction de la requête, nous vous invitons à utiliser la commande **`case`** que vous trouverez en annexe. Enfin, pour écrire une boucle infinie, il suffit d'écrire **`while true; do ... done`**.

Question 1.f. À terme, votre serveur est utilisé en parallèle par de nombreux utilisateurs. On souhaite donc rendre l'exécution des commandes parallèles. Il faut donc exécuter les commandes en arrière plan. Malheureusement, vous pourriez avoir des incohérences lorsque deux requêtes accédant aux fichiers ou répertoires du même utilisateur s'exécutent en parallèle.

Il faut donc modifier tous vos scripts pour éviter ces incohérences. Pour cela, vous devez utiliser un verrou nommé **`$id.lock`** lorsque vous accédez aux fichiers ou répertoires associés à l'utilisateur **`$id`**.

Modifiez **`mini-facebook.sh`** pour lancer les commandes en arrière plan et modifiez les quatre fichiers de commandes pour assurer la cohérence.

*Attention : Dans **`post-message.sh`**, les messages sont postés chez le récepteur et non chez l'émetteur. Il faut donc protéger le répertoire du récepteur.*

Partie 3 : Mise en place des clients

On souhaite maintenant mettre en place des clients pour le serveur.

Question 1.g. Écrivez un script **`client.sh id`** prenant en argument un identifiant **`$id`**. Ce script est une première ébauche de client. Il vérifie d'abord que l'argument a bien été donné. Ensuite, votre client exécute une boucle infinie. À chaque pas de la boucle, le client lit une requête sous la forme **`req args`**. Si **`$req`** n'est pas vide, le programme affiche alors sur l'écran la ligne **`req id args`**, où **`$id`** est l'identifiant passé en argument de **`client.sh`**.

Question 1.h. On souhaite maintenant connecter le client et le serveur avec des tubes nommés. Le serveur doit créer un tube nommé **`mini-facebook.pipe`**, alors que le client doit créer un tube nommé **`$id.pipe`**, dans lequel **`$id`** est l'identifiant passé en argument de **`client.sh`**.

De façon à éviter des ouvertures et fermetures de tube intempestives pouvant mener à des blocages ou des erreurs lorsque le tube est fermé alors qu'il reste un interlocuteur, vous devez impérativement utiliser des redirections avancées. On vous demande donc d'ajouter la commande suivante après votre création du tube :

```
exec 3<> nom-du-tube
```

où **`nom-du-tube`** est le nom du tube que vous venez de créer.

Modifiez vos scripts **`mini-facebook.sh`** et **`client.sh`** en conséquence.

***Remarque :** Les tubes sont juste crées dans cette question, ils seront utilisés plus tard dans l'exercice (questions j et k).*

Question 1.i. Que ce soit pour le serveur ou le client, le tube nommé doit être détruit lorsque le programme quitte. Actuellement, le programme quitte car vous utilisez vraisemblablement la combinaison de touche **`control-c`** pour envoyer un signal **`INT`** au processus. Dans le serveur et le client, vous devez donc intercepter ce signal pour détruire le tube et quitter le processus avec la commande **`exit 0`**. Modifiez vos scripts **`mini-facebook.sh`** et **`client.sh`** en conséquence.

Question 1.j. Au lieu d'afficher **`req id args`** sur le terminal, le client doit maintenant envoyer ses requêtes sur le tube du serveur. Le serveur, de son côté, doit lire les requêtes à partir de son tube nommé au lieu de les lire à partir du terminal. Modifiez **`client.sh`** et **`mini-facebook.sh`** en conséquence.

***Remarque :** Pour tester vos programmes, nous vous conseillons d'ouvrir deux terminaux, l'un pour exécuter un client, l'autre pour exécuter le serveur.*

Question 1.k. Au lieu d'afficher les réponses dans son terminal, votre serveur doit maintenant envoyer les réponses au client. Modifiez votre serveur pour que chacune des quatre commandes écrivent dans le tube du client au lieu d'écrire sur le terminal. Pensez à utiliser **`$cat $id.pipe`** pour vérifier que les données envoyées par le serveur à l'utilisateur **`$id`** sont bien écrites dans le tube associés à l'utilisateur **`$id`**.

Remarque : *Sur l'ensemble du code, vous n'avez à faire que quatre redirections.*

Question 1.l. Maintenant, on souhaite afficher le contenu du tube nommé chez le client. Le serveur peut envoyer trois types de messages différents, identifiés par le premier mot reçu :

- Le premier mot est égale à **`start-of-file`**. Dans ce cas, la réponse à afficher se trouve sur les lignes suivantes, jusqu'au **`end-of-file`**.
- Le premier mot est égal à **ok**. Dans ce cas, tout s'est bien passé et vous pouvez afficher un message comme **`Command successfully executed`**.
- Le premier mot est égal à **nok**. Dans ce cas, une erreur est apparue sur le serveur. Vous pouvez afficher un message comme **`Error: $msg`**, où **`$msg`** contient la fin de la ligne reçue (rappelez-vous que **`$msg`** stocke la fin de la ligne dans **`read ok msg`**).

Bravo, vous venez d'écrire votre premier serveur multiprocessus, bienvenue en système !