TD1 : Scripts bash

I. Courrier gagnant

1) Écrire un script `couga.sh` qui lit au clavier un nom, prénom, numéro de rue, nom de la rue, code postal et ville, puis affiche sur la sortie standard l'adresse complète de la personne.

2) Le script lit au clavier la somme d'argent gagnée. Si la somme est strictement positive, le script affiche un message du genre "Cher(e) X, vous avez gagné la somme de Y Euros". Si la somme est nulle, le message affiché sera du genre "Cher(e) X, vous n'avez pas gagné cette fois-ci". Pour toute autre valeur saisie, le message affiche un message d'erreur sur la sortie d'erreur et échoue.

3) Le script lit au clavier le nom d'un fichier, puis le crée, sinon il affiche une erreur et échoue.

4) Le script écrit le courrier (adresse et message de gain ou de perte) dans le fichier.

II. Opacification de texte

1) Écrire un script bash `opac.sh` qui demande à lire un texte sur l'entrée standard, jusqu'à la fin de l'entrée standard (`^D`). Le script affiche ensuite le texte en inversant l'ordre des lignes.

2) Le script demande un nom de fichier en entrée et un nom de fichier en sortie. Il vérifie qu'il peut lire le fichier en entrée et créer le fichier en sortie. Enfin il lit le fichier en entrée et le recopie dans le fichier en sortie en inversant l'ordre des lignes.

3) Le script transforme chaque ligne du texte en décalant les lettres : `'a'` ↔ `'n'`, `'b'` ↔ `'o'`, etc (opération appelée "rot13") en utilisant la commande `tr`.

Rappels

La commande `test` admet principalement les options suivantes :

`-d fichier` vrai si le fichier est un répertoire.
`-e fichier` vrai si le fichier existe.
`-f fichier` vrai si le fichier est régulier.
`-s fichier` vrai si le fichier est non vide.
`-r fichier` vrai si vous pouvez lire le fichier.
`-w fichier` vrai si vous pouvez écrire dans le fichier.
`-x fichier` vrai si vous pouvez exécuter le fichier.
`fichier1 -nt fichier2` vrai si fichier1 est plus récent que fichier2 (en date de modification).
`fichier1 -ot fichier2` vrai si fichier1 est plus ancien que fichier2.
`-z chaîne` vrai si la chaîne est vide.
`-n chaîne` vrai si la chaîne est non vide.
`chaîne1 = chaîne2` vrai si les chaînes sont égales.
`chaîne1 != chaîne2` vrai si les chaînes sont différentes.
`chaîne1 \< chaîne2` vrai si chaîne1 est inférieure à chaîne2 dans l'ordre lexicographique.
`chaîne1 \> chaîne2` vrai si chaîne1 est supérieure à chaîne2.
`\( expression \)` vrai si l'expression est vraie.
`! expression` vrai si l'expression est fausse.
`expression1 -a expression2` vrai si les deux expressions sont vraies.
`expression1 -o expression2` vrai si l'une au moins des expressions est vraie.
`arg1 OP arg2` Tests arithmétiques, où `OP` est `-eq`, `-ne`, `-lt`, `-le`, `-gt`, ou `-ge`.

TP1 : Scripts bash

I. Une ligne sur deux

Écrire un script `ligpa.sh` qui demande un nom de fichier en entrée. Il vérifie qu'il peut lire le fichier en entrée, sinon affiche une erreur et échoue.

Le script affiche sur la sortie standard toutes les lignes de numéro pair, ceci grâce à une variable que l'on met alternativement aux valeurs `"impair"` ou `"pair"`.

Créer un fichier texte comprenant au moins une dizaine de lignes différentes pour faire des tests (par exemple des lignes commençant par `"un .."`, `"deux .."`, ..). N'oubliez pas de mettre les droits d'exécution au script (`chmod +x ligpa.sh`) avant de le lancer (`./ligpa.sh`) dans le terminal.

II. Début et fin d'un fichier texte

1) Écrire un script `bazar.sh` qui demande un nom de fichier en entrée et un nom de fichier en sortie. Il vérifie qu'il peut lire le fichier en entrée et créer le fichier en sortie.

2) Le script demande un entier `a` et un entier `b`. Il vérifie que $0 \leqslant$ `a` $\leqslant$ `b`, sinon il affiche un message d'erreur et échoue.

3) Le script écrit dans le fichier de sortie les lignes numéro `a` à `b` du fichier d'entrée en se servant des commandes `tail` et `head` reliées par des tubes. Tester sur le fichier d'exemple de l'exercice précédent.

4) Dans votre script, commentez la partie concernant la question précédente ; modifier de façon à ce que le script écrive dans le fichier de sortie toutes les lignes du fichier d'entrée sauf les lignes numéro `a` à `b`, en se servant des commandes `tail` et `head`.

Pour calculer `a-1` ou `b+1` vous pouvez utiliser la commande `expr` redirigée dans un fichier temporaire, dont vous lisez ensuite la première ligne avec `read`.

N'oubliez pas de supprimer les fichiers temporaires avec la commande `rm`.

5) Le script doit maintenant recopier le fichier d'entrée vers le fichier de sortie, de manière à ce que les lignes apparaissent dans l'ordre originel, sauf les lignes numéro `a` à `b` qui apparaîtront triées dans l'ordre lexicographique décroissant (commande `sort`).

Rappels

- ▷ La commande `expr` affiche le résultat d'un calcul passé en argument ; par exemple `expr 4 + 8` affiche `12`. Attention aux espaces.
- ▷ Le shell substitue `$$` par le numéro de processus du shell ; on peut s'en servir pour créer un nom de fichier temporaire unique (par exemple `"tmp-$$.txt"`).
- ▷ `echo -n` affiche une ligne sans retour chariot ; `echo -e` interprète les `"\n"`.
- ▷ `head -`k `[fichier]` affiche les k premières lignes (par défaut $k = 10$) du fichier passé en paramètre (sinon de l'entrée standard).
- ▷ `tail -`k `[fichier]` affiche les k dernières lignes (par défaut $k = 10$) du fichier passé en paramètre (sinon de l'entrée standard). La version `tail -n +`k imprime de la ligne k à la fin.
- ▷ `sort [options]` trie les lignes lues sur l'entrée standard et les recopie sur la sortie standard, selon `option` (par défaut dans l'ordre lexicographique croissant) :
  - `-n` dans l'ordre de la valeur numérique du premier mot ;
  - `-r` dans l'ordre décroissant.

TD2 : Scripts bash

I. Motifs

1) Écrivez le script `casli.sh` qui lit l'entrée standard, et affiche à l'aide de l'instruction `case` les lignes contenant le mot `"bonjour"`.

2) Rediriger l'affichage dans un fichier temporaire, puis afficher le nombre de lignes contenant le mot `"bonjour"` à l'aide de la commande `wc`.

3) Procéder de même sans fichier temporaire avec un tube.

4) Modifiez le script `casli.sh` pour qu'il affiche le nombre de lignes contenant le mot `"bonjour"` ou `"Bonjour"` sans contenir le mot `"salut"`.

II. Arguments et boucles

1) Écrire le script `calsop.sh` qui vérifie la présence de 3 arguments, sinon affiche le message d'erreur `"Usage: calsop.sh somme|produit a b"` et échoue.

2) Le script vérifie que le premier argument est valide (c'est-à-dire d'après l'usage qu'il vaut `"somme"` ou `"produit"`) sinon affiche une erreur et échoue.

3) Le script calcule et affiche la somme ou le produit de `a` et `b` en se servant de `expr`.

4) On modifie l'usage : `"calsop.sh somme|produit entier ..."`. Le script calcule la somme ou le produit des entiers en argument à l'aide d'une boucle `while`.

5) Même question avec une boucle `for`.

Rappels

- ▷ La commande `wc -l` compte le nombre de lignes lues sur l'entrée standard puis l'affiche sur la sortie standard.
- ▷ La commande `expr a op b` où `op` est `+` ou `'*'` affiche le résultat du calcul.

TP2 : Scripts bash

I. Oui ou non ?

1) Écrivez le script `ligon.sh` qui vérifie la présence d'au moins un argument, sinon affiche un message d'erreur puis échoue.

2) Le script affiche chaque argument sur une ligne, suivi de `"(O/N) ?"` puis lit la réponse de l'utilisateur. Par exemple,

```
$ ./ligon.sh un deux trois quatre
un (O/N) ? O
deux (O/N) ? N
trois (O/N) ? N
quatre (O/N) ? O
```

3) Le script enregistre les arguments pour lesquels la réponse est `"O"` dans un fichier temporaire, puis affiche le fichier temporaire et le supprime. Dans notre exemple on verra donc :

```
un
quatre
```

II. Nouvelles

1) Écrire un script `news.sh`. Le script vérifie la présence du répertoire `nouvelles` sinon il le crée.

2) Le script affiche la liste des fichiers contenus dans le répertoire `nouvelles`, avec un fichier par ligne.
Pour tester dans le terminal, créer des fichiers vides en vous servant d'une expansion d'accolades, avec des noms comportant des espaces.

3) Créer dans le répertoire `nouvelles` un fichier caché `.temoin`, puis créer quelques fichiers supplémentaires (qui seront donc plus récents que `.temoin`).

Modifier le script afin qu'il n'affiche que les noms des fichiers qui sont plus récents que `.temoin`.

4) Le script accepte un ou plusieurs arguments parmi `-liste`, `-lire`, `-toutlu`. Si aucun argument n'est présent, ou si un argument n'est pas dans la liste, le script affiche l'usage et échoue.

5) Si l'argument `-liste` est présent, le script affiche les noms des fichiers qui sont plus récents que `.temoin`. Si l'argument `-lire` est présent, le script affiche le nom et le contenu de chaque fichier qui est plus récent que `.temoin`. Si l'argument est `-toutlu`, le script met `.temoin` à la date courante.

6) Rajouter une option `-purge` qui supprime les nouvelles plus anciennes que `.temoins`.

TD3 : Scripts bash

I. Photos favorites

1) Écrire la fonction `afficher_usage` qui affiche sur la sortie standard le message
`"Usage: $0 [-h|--help] [-t|--taille] [-f|--favo fichier] photo ..."` .

2) Écrire le script `favopic.sh` qui teste la présence d'au moins un argument, sinon il affiche l'usage sur la sortie d'erreur et échoue.

3) Écrire la fonction `afficher_nom_photo` qui prend en argument un chemin de fichier de photo. La fonction lui enlève le chemin et les extensions puis affiche le nom de la photo sur la sortie standard.

4) On suppose disposer d'un fichier des photos favorites, constitué de noms de photos (sans chemin ni extension), un nom par ligne.

Écrire la fonction `est_favorite` qui prend en argument le fichier des photos favorites et un chemin de fichier de photo. La fonction réussit si la photo est une photo favorite.

5) On suppose que dans le script sont définies les variables globales `flag_taille`, `flag_favorites` et `fichier_favorites`.

Écrire la fonction `afficher_proprietes_photo` qui prend en argument un nom de chemin de photo. Elle affiche sur une même ligne le nom de la photo, sa taille si `flag_taille` est vrai, un `"F"` si `flag_favorites` est vrai et si la photo est une favorite.

6) Écrire la fonction `lister_photos` qui prend en argument des chemins de fichiers de photos. La fonction vérifie pour chaque fichier (avec une boucle `for`) s'il existe ; dans ce cas elle affiche les propriétés de la photo, sinon elle affiche le chemin du fichier dans un message d'erreur.

7) Écrire le corps du script d'après l'usage donné ci-dessus. Le script crée les variables globales `flag_taille`, `flag_favorites` et `fichier_favorites` selon la valeur des options présentes, vérifie si besoin l'existence de `fichier_favorites`, puis appelle `lister_photos` en lui passant les photos en argument.

Rappels

- ▷ La commande `grep -w mot_clé fichier` affiche les lignes du `fichier` contenant le `mot_clé` non accolé à une autre chaîne. Elle réussit si elle trouve au moins une ligne le contenant.
- ▷ La commande `wc -c < fichier` affiche la taille du `fichier`.

TP3 : Scripts bash

I. Script de compilation

On se propose d'écrire un script qui servira à compiler des fichiers C.

1) Créer trois fichiers `f1.c`, `f2.c` et `f3.c`, qui affichent leur nom dans la sortie standard (avec un `printf`). Les compiler avec `gcc` et tester les exécutables `f1`, `f2` et `f3`.

2) Créer un script `compc.sh` dans lequel une fonction `afficher_usage` affiche dans la sortie standard les différentes utilisations du script, qui sont :

```
compc.sh -h|--help
compc.sh --touch|--clean fichier.c ...
compc.sh [option ...] --cc fichier.c ...
```

où `option` est `--debug|--optim|--warni`

3) Le script teste la présence d'au moins un argument, sinon il affiche l'usage sur la sortie d'erreur et échoue.

4) Décoder dans le script les arguments de la ligne de commande, en vous inspirant de la méthode utilisée à la fin du TD3 avec un `case` et des flags. On peut simplifier le procédé en mémorisant une variable `action` parmi `-h|--help|--touch|--clean|--cc` et en fixant des flags uniquement pour les options. Pour tester, afficher provisoirement leurs valeurs après la phase de décodage.

5) Écrire une fonction `modifier_date_fichiers` qui reçoit des noms de fichiers C en argument. Pour chaque fichier, la fonction vérifie son existence, le met à la date courante et affiche "fichier : date changée", sinon affiche le message d'erreur "fichier : n'existe pas".
Appeler `modifier_date_fichiers` pour l'action `--touch` du script et tester.

6) Écrire une fonction `obtenir_nom_executable` qui prend en argument le nom d'un fichier C. La fonction affiche le nom de l'exécutable en enlevant l'extension du fichier.

7) Écrire une fonction `nettoyer_fichiers` qui reçoit des noms de fichiers C en argument. Pour chaque fichier, la fonction vérifie son existence (sinon affiche l'erreur "fichier : n'existe pas"), obtient le nom de l'exécutable correspondant (par substitution de commandes), affiche "exécutable : suppression" et l'efface s'il existe, sinon affiche "exécutable : absent, non supprimé".
Appeler `nettoyer_fichiers` pour l'action `--clean` du script et tester.

8) Écrire une fonction `compiler_fichiers` qui reçoit des noms de fichiers C en argument. Pour chaque fichier, la fonction vérifie son existence (sinon affiche "fichier : n'existe pas"), obtient le nom de l'exécutable correspondant, puis compile le fichier C avec `gcc`, en lui fournissant selon les flags les options nécessaires (`-g` ou `-O2` ou `-Wall -W`, voir rappels) et en rajoutant `-std=c99`.

La fonction ne compile que les fichiers C qui n'ont pas déjà été compilés ou qui sont plus récents que l'exécutable correspondant (sinon message "fichier : à jour").

Appeler `compiler_fichiers` pour l'action `--cc` du script et tester.

Rappels

- La commande `gcc [option ...] fich.c -o fich` compile un fichier `fich.c` en un exécutable `fich`. Si l'option `-g` est présente, `gcc` rajoute des informations de déboguage dans l'exécutable, qui seront utiles pour le débogueur `gdb`. Si l'option `-O2` est présente, `gcc` optimise le code produit. Si les options `-Wall -W` sont présentes, `gcc` affiche des warnings.

TD4 : Scripts bash

I. Parcours récursif de sous-répertoires

Exercice à faire avec des motifs, sans utiliser les commandes `cd`, `ls`, `find`.

1) Écrire la fonction `lister_repertoire` qui reçoit en argument un chemin de répertoire `rep`. La fonction affiche récursivement le chemin des fichiers réguliers et des sous-répertoires situés dans `rep`.

2) Écrire la fonction `chercher_fichier` qui prend en argument un chemin de répertoire `rep` et un nom de fichier `nomf` sans chemin. La fonction parcourt récursivement `rep` et ses sous-répertoires, et affiche le chemin de toutes les occurrences de `nomf`.

3) Écrire le script `rls.sh` dont l'usage est : `rls.sh [--find fichier] [répertoire]`. Si l'option `--find` est présente, le script appelle `chercher_fichier` pour le `fichier`, sinon il appelle `lister_repertoire`. Si le répertoire n'est pas donné en argument on utilise le répertoire courant.

II. Expansion arithmétique

1) Écrire le script `mkfiles.sh [n] prefixe` qui créé n fichiers vides (à défaut, 5) de la forme `prefixe.1`, `prefixe.2`, ..., `prefixe.`n.

2) On dispose d'un fichier texte faisant l'inventaire de fruits. Chaque ligne est sous la forme `"fruit [:] nombre qualité"`, où `fruit` est un mot, `nombre` un entier et `qualité` est un ou plusieurs mots ; le nombre de blancs est quelconque.

a) Écrire la fonction `compter_fruits fruit fichier` qui affiche la somme des nombres de `fruit` présents dans le `fichier`.

b) Écrire le script `fruits.sh fichier fruit ...` qui pour chaque fruit en argument affiche la somme des nombres de `fruit` présents dans le `fichier`, puis affiche le fruit majoritaire.

TP4 : Scripts bash

I. Sauvegardes numérotées

On se propose d'écrire un script `numsauv.sh` qui effectue des copies numérotées de chaque fichier donné en argument dans un sous-répertoire `svg`. Le script peut aussi supprimer toutes les copies sauf celle dont le numéro est le plus grand (option `--purge`).

1) Écrire la fonction `afficher_usage` qui affiche dans la sortie standard l'usage du script :
`USAGE: numsauv.sh [--purge] fichier ...`

2) Le script teste la présence d'au moins un argument, sinon il affiche l'usage sur la sortie d'erreur et échoue.

3) Déterminer dans le script si celui-ci est invoqué avec l'argument `--purge` .

4) Écrire la fonction `verifier_repertoire_svg` qui créer automatiquement le sous-répertoire `svg` si celui-ci n'existe pas déjà. Appeler la fonction dans le script.

5) Écrire la fonction `extraire_extension` qui prend en argument un chemin de fichier et affiche la dernière extension sans le point.

6) Écrire la fonction `est_entier` qui réussit si l'argument est entier.

7) Écrire une fonction `chercher_copie_max` qui reçoit en argument un nom de fichier `nomf` du répertoire courant ; elle cherche le maximum des entiers k pour les fichiers `svg/nomf.`k (0 par défaut), puis l'affiche sur la sortie standard. Tester.

Une solution consiste à parcourir la liste des fichiers `svg/nomf.*` et pour chacun, extraire l'extension, vérifier que c'est un entier puis le comparer au maximum en cours.

8) Écrire une fonction `sauvegarder_fichier` qui reçoit en argument un fichier `nomf` du répertoire courant et un entier `max`. La fonction compare ensuite le contenu du fichier `nomf` avec celui de `svg/nomf.max` (commande `cmp`, voir rappels). S'ils sont égaux, la fonction affiche

`== nomf identique a svg/nomf.max`

sinon, elle affiche

`Copie de nomf --> svg/nomf.`k

où k =`max`+1, et elle recopie le fichier `nomf` vers `svg/nomf.`k .

9) Rajouter à la fin du script une boucle qui, pour chaque fichier en argument du script, appelle `chercher_copie_max`, puis `sauvegarder_fichier` si l'argument `--purge` était absent. Tester.

10) Écrire une fonction `purger_copies` qui reçoit en argument un fichier `nomf` du répertoire courant et un entier `max`. La fonction supprime toutes les copies `svg/nomf.`k où $k <$ `max`. Brancher dans le script dans le cas où l'argument `--purge` était présent. Tester.

Rappels

- ▷ La commande `cmp fichier1 fichier2` réussit si les deux fichiers ont exactement le même contenu. Dans le cas contraire elle affiche la présence de différences ou des messages d'erreur.
- ▷ La commande `expr 1 + x` échoue si `x` n'est pas un entier.

TD5 : Scripts bash et tableaux

I. Pgcd

1) Écrire une fonction récursive `calculer_pgcd a b` qui calcule le plus grand commun diviseur entre 2 nombres `a` et `b` positifs puis l'affiche sur la sortie standard. Utiliser l'algorithme suivant :

$$\text{pgcd}(a,b) = \begin{cases} a+b & \text{si } a*b=0 \\ \text{pgcd}(a, b \bmod a) & \text{si } a \leq b \\ \text{pgcd}(a \bmod b, b) & \text{si } a > b \end{cases}$$

On obtient par exemple : $\text{pgcd}(85,25) = \text{pgcd}(10,25) = \text{pgcd}(10,5) = \text{pgcd}(0,5) = 5$.

2) Écrire une fonction `calculer_pgcd_tab` qui calcule et affiche le pgcd des nombres positifs stockés dans un tableau global tassé `tab`. Utiliser la propriété : $\text{pgcd}(a,b,c) = \text{pgcd}(\text{pgcd}(a,b),c)$.

3) Écrire une fonction `calculer_abs_tab` qui remplace dans le tableau global tassé `tab` les entiers négatifs par leur valeur absolue.

4) Écrire un script `pgcd.sh x1 .. xn` qui calcule le pgcd des entiers passés en argument puis l'affiche sur la sortie standard.

II. Tableaux associatifs

Exercice à faire sans utiliser la commande `join`.

1) Écrire la fonction `memoriser_fichier fichier tab` où `fichier` est le nom d'un fichier texte et `tab` est un tableau associatif passé par référence. On considère pour chaque ligne que le premier mot est la `clé`, et que le reste de la ligne constitue la `valeur`.

La fonction lit le fichier et mémorise chaque couple `clé`, `valeur` dans le tableau associatif `tab`.

Dans le cas d'une erreur de lecture elle échoue et affiche un message d'erreur.

2) Écrire la fonction `joindre_tableaux tab1 tab2 tabres` où `tab1`, `tab2` et `tabres` sont des tableaux associatifs passés par référence.

La fonction fait la jointure de `tab1` et `tab2` dans `tabres` ; autrement dit, l'ensemble des clés de `tabres` est l'union des clés des 2 tableaux, tandis que les valeurs dans `tabres` sont la concaténation des valeurs correspondantes dans les 2 tableaux (séparées par un blanc).

Exemple :

```
tab1=([lundi]="steak" [mardi]="poisson pané")
tab2=([jeudi]="soupe" [lundi]="frites" )
```

Le résultat attendu est

```
tabres=([lundi]="steak frites" [mardi]="poisson pané" [jeudi]="soupe")
```

3) Écrire un script de test.

TP5 : Scripts bash et tableaux

I. Recherche de fichiers identiques

On se propose de réaliser un script bash `doublons.sh` qui trouve les fichiers au contenu identique dans une hiérarchie de répertoires en comparant les sommes MD5 des fichiers. La somme MD5 est le résultat d'une fonction de hachage très complexe sur le contenu du fichier, présentée sous forme alphanumérique. Il est quasiment impossible de trouver deux fichiers différents qui aient la même somme MD5, et donc, si deux fichiers ont la même somme MD5, alors on peut affirmer qu'ils sont identiques. Pour calculer la somme MD5 d'un fichier on utilise la commande `md5sum` :

```
$ md5sum tp05.tex
8beaa58a83d0f2b9662b287628288c4a  tp05.tex
```

1) Écrire la fonction `calculer_sommes_md5 rep` qui reçoit en paramètre un répertoire `rep`. La fonction parcourt récursivement les fichiers et sous-répertoires de `rep`. Pour chaque fichier rencontré la fonction affiche sur une même ligne la somme MD5, du blanc puis le chemin du fichier.

2) Écrire la fonction `memoriser_sommes_md5 tab_fic` qui reçoit par référence un tableau associatif `tab_fic`. La fonction commence par vider `tab_fic` ; puis elle lit sur l'entrée standard une série de lignes au format `"somme_md5 chemin_fichier"`, et stocke au fur et à mesure dans `tab_fic` la valeur `somme_md5` associée à la clé `chemin_fichier`.

Remarque : vérifiez votre version de bash en tapant : `bash --version`. Si votre version de bash est antérieure à 4.3, alors au lieu de passer les tableaux par référence il faudra les déclarer en global.

3) Écrire la fonction `detecter_doublons tab_fic tab_cpt tab_dbl` qui reçoit par référence trois tableaux associatifs `tab_fic`, `tab_cpt` et `tab_dbl`. Le tableau en entrée `tab_fic` contient des sommes MD5 en valeur, associées à des chemins de fichiers en clés. Les tableaux en sortie `tab_cpt` et `tab_dbl` associeront quant à eux des sommes MD5 en clés, aux valeurs suivantes :

- ▷ le nombre de fichiers qui possèdent cette somme MD5 pour `tab_cpt` ;
- ▷ la liste des chemins de fichiers, séparés par `":"`, qui possèdent cette somme MD5 (et qui sont donc des doublons), pour `tab_dbl`.

Les tableaux en sortie sont d'abord vidés. Ensuite on parcourt chaque élément de `tab_fic` et on teste si la valeur somme MD5 existe déjà en tant que clé dans `tab_cpt`. Si c'est non, on insère le chemin de fichier en valeur pour la clé somme MD5 dans `tab_dbl`, et on insère 1 en valeur pour la même clé dans `tab_cpt`. Si c'est oui, on incrémente la valeur de la case de `tab_cpt` associée à la clé somme MD5, et on concatène un `":"` puis le chemin du fichier à la valeur actuelle de la case de `tab_dbl` associée à la clé somme MD5.

4) Écrire la fonction `afficher_doublons tab_cpt tab_dbl` qui reçoit par référence deux tableaux associatifs `tab_cpt` et `tab_dbl`. La fonction affiche tous les doublons contenus dans les deux tableaux, c'est-à-dire tous les fichiers dont la valeur dans `tab_cpt` est > 1. Le format d'affichage est constitué d'une ligne par liste de doublons : le nombre d'occurrences, puis du blanc, puis les chemins des fichiers doublons séparés par des `":"`.

5) Écrire le programme principal, dont l'usage est `doublons.sh rep`. Le script vérifie la présence d'un argument, sinon affiche l'usage sur la sortie d'erreur et échoue. On déclare ensuite trois tableaux associatifs `arr_fic`, `arr_dbl` et `arr_cpt`.
Le script affiche `"Calcul des sommes md5 ..."`, puis appelle `calculer_sommes_md5` pour `rep`, et mémorise au fur et à mesure les résultats affichés dans `arr_fic` grâce à `memoriser_sommes_md5` et un *retournement de tube* (vu en cours).
Le script affiche ensuite `"Détection des doublons ..."` puis détecte les doublons en appelant `detecter_doublons`. Le script affiche enfin `"Affichage des doublons ..."` puis affiche les doublons en appelant `afficher_doublons`.

TD6 : Création de processus en C

I. Création

1) Écrire un programme C qui affiche son PID, puis crée deux fils. Ils affichent leur PID et leur filliation et se terminent.

2) Écrire un programme C qui affiche son PID, puis crée un fils et un petit-fils. Ils affichent leur PID et leur filliation et se terminent.

3) Écrire un programme C qui change de PID toutes les 2 secondes, et affiche à chaque fois celui-ci.

II. Terminaison

1) Écrire un programme C qui affiche son PID, puis crée un fils. Le père affiche son PID, puis affiche un message signalant que son fils est mort lorsque c'est bien le cas. Le fils affiche son PID, s'endort 5 secondes puis se termine.

2) Écrire un programme C qui affiche son PID, puis crée un fils. Le père affiche son PID, attend la terminaison du fils puis affiche son code de sortie.

III. Recouvrement

Écrire un programme `verifier.c` dont l'usage sera `verifier com arg1 .. argn`, et qui effectue les opérations suivantes :

1) Il lance la commande `com arg1 .. argn` et signale une éventuelle erreur lors du lancement.

2) Il attend la fin de l'exécution et précise par un message le résultat (succés ou échec).

Rappels

- ▷ `sleep(n);` endort un processus pendant `n` secondes.

TP6 : Création de processus en C

I. Création de fils

1) Écrire un programme C qui crée 5 fils. Le programme et chacun des fils affiche son PID et se termine.

2) Modifier le programme pour que le nombre de fils qu'il crée soit reçu en argument de la ligne de commande.

3) Modifier le programme pour que chaque fils affiche de plus son rang dans la fratrie.

II. Création de petits-fils

1) Écrire un programme C qui crée 3 fils, chacun des fils créant 2 petits-fils. Le programme et chacun des fils ou petits-fils affiche son rang dans la fratrie, son PID et se termine.

2) Modifier le programme pour que le nombre de fils et de petits-fils qu'il crée soient reçus en argument de la ligne de commande.

III. Recouvrement séquentiel

Écrivez un programme C qui, par le biais de créations de processus et de recouvrements, exécute la suite de commandes `who ; pwd ; ls -l` (rappel : le point-virgule signifie qu'une commande est exécutée lorsque la commande précédente est terminée).

IV. La commande `myif`

Écrire un programme C `myif.c` qui admet la ligne de commande suivante :

`myif` *commande1 args ...* `--then` *commande2 args ...* [`--else` *commande3 args ...*] `--fi`

Le programme exécute la première commande puis, selon qu'elle aie réussi ou échoué, exécute la deuxième ou la troisième commande.

Rappels

- La fonction `int atoi(char *s);` de `stdlib.h` convertit le début de la chaîne pointée par `s` en entier de type `int`.

TD7 : Recouvrements et signaux en C

I. Recouvrement en sursis

Écrire un programme `atkill.c` dont l'usage est `atkill nbsec sig com [arg ...]` et qui effectue les opérations suivantes :

1) Il lance la commande `com [arg ...]` et signale une éventuelle erreur lors du lancement.

2) Il provoque l'envoi au bout de `nbsec` secondes du signal `sig` à la commande puis attend sa terminaison.

II. La commande `mywhile`

On se propose d'écrire un programme C `mywhile.c` qui admet la ligne de commande suivante :
`mywhile commande1 [arg ...] --do commande2 [arg ...] --done`

1) Écrire la fonction `int indice_mot (int argc, char *argv[], char *mot)` qui reçoit en paramètres le nombre `argc` et le tableau `argv` des arguments de la ligne de commande, et une chaîne de caractères `mot`. La fonction renvoie l'indice de la première occurence de `mot` dans `argv`, ou `-1` sinon.

2) Écrire la fonction `void execute_com (char *argv[], int ind1, int ind2)` qui reçoit en paramètres le tableau `argv` des arguments de la ligne de commande, et deux entiers `ind1` et `ind2` tels que $0 <$ `ind1` $<$ `ind2` $\leq$ `argc`. La fonction effectue un recouvrement avec la commande située à l'indice `ind1` dans `argv`, munie de ses arguments suivants dans `argv` ; `ind2` est l'indice qui suit le dernier argument. Si le recouvrement échoue, la fonction termine le programme.

3) Écrire le programme principal, qui recherche l'indice des arguments `--do` et `--done`, puis affiche l'usage du programme et le termine si la syntaxe n'est pas respectée.
Ensuite, le programme exécute `commande1` et attend sa fin. Si elle a échoué, le programme se termine, sinon il exécute `commande2` et attend sa fin, puis recommence à exécuter `commande1`, etc.
Lorsque le programme se termine, il renvoie le *status* de la dernière exécution de `commande2`, à défaut `1`. Pour chaque exécution de commande, le programme se duplique et le fils se recouvre.

Rappels

- ▷ `sleep(n)` endort un processus pendant `n` secondes.

TP7 : Recouvrements et signaux en C

I. Capture des signaux en C

1) Recopier le programme `testsig.c` suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
    while (1) { sleep (1); printf ("Alive !\n"); }
}
```

Compiler avec `gcc -W -Wall -std=c99 testsig.c -o testsig`; exécuter, puis au bout de quelques secondes interrompre en tapant `^C`.

2) Nous allons protéger le programme en captant le signal `SIGINT`. Rajouter dans le programme :

```
#include <signal.h>
void capter_INT ()
{
    printf ("Ouch !\n");
}
```

puis au début du `main` rajouter : `signal (SIGINT, capter_INT);` Recompiler, exécuter, interrompre au bout de quelques secondes en tapant `^C`; réessayer une seconde fois. Moralité ?

3) Rendre le programme impossible à interrompre par `^C`.

4) Capter le signal `SIGALRM` avec une fonction qui affiche un message et termine le programme. Utiliser la fonction `alarm()` pour que le programme s'arrête au bout de 10 secondes.

5) Modifier le programme de façon à ce qu'il s'arrête au bout de 5 secondes après la frappe du dernier `^C`.

II. La commande `mytime`

Écrire un programme C `mytime.c` qui admet la ligne de commande suivante :

```
mytime [-n k] [-s] commande [arg ...]
```

Le programme exécute la `commande` avec ses arguments éventuels, attend sa terminaison puis affiche la durée d'exécution de la commande en secondes et microsecondes.

Si l'option `-n` est présente, la même chose est faite `k` fois (`k` > 0) ; de plus la durée moyenne est affichée. Si l'option `-s` est présente, le programme affiche également chaque fois le code de sortie de la commande.

Pour mesurer la durée, on peut se servir de la fonction `gettimeofday()` qui fournit le nombre de secondes et microsecondes écoulées depuis l'Epoch (le 1^{er} janvier 1970 à 0h) ; il suffit de l'appeler avant et après l'exécution puis de calculer la différence. Les fichiers `sys/time.h` et `time.h` définissent `gettimeofday(struct timeval *tv, NULL)` et `struct timeval { time_t tv_sec, tv_usec;}` où les champs sont des entiers et contiennent le nombre de secondes et microsecondes.

Rappels

- La fonction `alarm(unsigned int n);` de `unistd.h` provoque l'envoi d'un signal `SIGALRM` au processus en cours au bout de `n` secondes. Chaque nouvel appel à `alarm` annule et remplace le précédent.

TD8 : Signaux Unix

I. Alarme

La fonction `signal` n'étant pas portable, on se donne une fonction `mysignal` qui simplifie l'usage de `sigaction` pour placer un handler de signal :

```
#define _GNU_SOURCE  // en premier
#include <signal.h>

/*  Place le handler de signal void h(int) pour le signal sig avec sigaction().
    Le signal est automatiquement masqué pendant sa délivrance.
    Si options = 0,
      - les appels bloquants sont interrompus avec retour -1 et errno = EINTR.
      - le handler est réarmé automatiquement après chaque délivrance de signal.
    si options est une combinaison bit à bit (opérateur |) de
      - SA_RESTART : les appels bloquants sont silencieusement repris.
      - SA_RESETHAND : le handler n'est pas réarmé.
    Renvoie le résultat de sigaction.
*/

int mysignal (int sig, void (*h)(int), int options)
{
    struct sigaction s;
    s.sa_handler = h;
    sigemptyset (&s.sa_mask);
    s.sa_flags = options;
    int r = sigaction (sig, &s, NULL);
    if (r < 0) perror (__func__);
    return r;
}
```

1) Écrire un programme C qui recopie l'entrée standard sur la sortie standard, caractère par caractère. Toutes les 2 secondes, le programme incrémente un compteur et l'affiche ; il se termine au bout de 5 incrémentations.

2) Écrire un programme C qui crée un fils. Le père recopie l'entrée standard sur la sortie standard, caractère par caractère. Le fils vérifie chaque seconde l'existence du père, et affiche un message et se termine lorsqu'il a détecté la fin du père.

II. Transmission de valeur

1) Écrire le programme `sig-send.c` dont l'usage est `sig-send sig val pid`, et qui envoie le signal `sig` et la valeur entière `val` au processus de PID `pid`.

2) Écrire le programme `sig-recv.c` qui met en place un handler pour tous les signaux captables. Pour chaque signal capté, le programme affiche le PID de l'émetteur, et si une valeur est associée au signal, elle est affichée.

Rappels

- ▷ La fonction `alarm(unsigned int n);` de `<unistd.h>` provoque l'envoi d'un signal `SIGALRM` au processus appelant au bout de `n` secondes.
- ▷ La fonction `kill(pid_t pid, int sig)` appelée avec `sig=0` n'envoie pas de signal mais renvoie 0 si le processus `pid` existe, sinon renvoie -1.

TP8 : Signaux Unix

I. Comptage de signaux

1) Écrire un programme C qui crée 1 fils, puis attend la fin du fils. Le fils envoie le signal `SIGUSR1` au père puis se termine.

2) Modifier le père pour qu'il affiche un message lorsque le signal `SIGUSR1` est capté.

3) Modifier le programme pour qu'il reçoive un entier n sur la ligne de commande, et modifier le fils de telle sorte qu'il envoie à la suite n signaux `SIGUSR1` au père avant de se terminer.

4) Modifier le père pour qu'il compte le nombre de signaux `SIGUSR1` reçus ; supprimer tout affichage dans le handler de signal du père, et afficher le nombre final de signaux reçus après la fin du fils.

5) Tester avec $n = 10,\ 1\,000,\ 100\,000$. Que peut-on conclure ?

II. Ping-pong de signaux

1) Écrire un programme C qui crée 1 fils, puis attend la fin du fils. Le fils envoie le signal `SIGUSR1` au père. À la réception du signal, le père affiche un message puis envoie le signal `SIGUSR1` au fils. À la réception du signal, le fils affiche un message puis se termine. Le père attend la fin du fils, affiche un message puis se termine.

2) Modifier le programme pour qu'il reçoive un entier n sur la ligne de commande, et modifier le fils et le père pour que l'échange de signaux se passe successivement n fois.

3) Modifier le programme en utilisant la fonction `sigqueue` et le flag `SA_SIGINFO`, de manière à supprimer toute variable globale : chaque processus renvoie le signal au processus émetteur (champ `si_pid)`, et décrémente l'entier associé au signal (champ `si_value`). Astuce : démarrer le compteur à `2n`.

Rappels

La fonction `signal` n'étant pas portable, on se donne une fonction `mysignal` qui simplifie l'usage de `sigaction` pour placer un handler de signal ; recopier la fonction vue en TD.

TD9 : Makefile

I. Makefile de compilation

1) Écrire l'arbre des dépendances des fichiers C suivants :

```
/* f1.h */
#ifndef F1_H
#define F1_H
typedef struct { int a,b; } T;
void put (T *t, int x);
int get (T *t);
void exch (T *t);
#endif /* F1_H */

/* f1.c */
#include "f1.h"
void put (T *t, int x) {
    t->a = x;
}
int get (T *t) {
    return t->a;
}
void exch (T *t) {
    int x = t->a;
    t->a = t->b; t->b = x;
}

/* f2.h */
#ifndef F2_H
#define F2_H
void affi_i (int i);
#endif /* F2_H */

/* f2.c */
#include <stdio.h>
#include "f2.h"
void affi_i (int i) {
    printf ("%d\n", i);
}
```

```
/* f3.h */
#ifndef F3_H
#define F3_H
#include "f1.h"
void affi_t (T *t);
#endif /* F3_H */

/* f3.c */
#include "f2.h"
#include "f3.h"
void affi_t (T *t) {
    affi_i (get (t));
    exch (t);
    affi_i (get (t));
    exch (t);
}

/* f4.h */
#ifndef F4_H
#define F4_H
int min (int a, int b);
#endif /* F4_H */

/* f4.c */
#include "f4.h"
int min (int a, int b) {
    return (a<b) ? a:b;
}
```

```
/* prog1.c */
#include "f1.h"
#include "f3.h"
int main () {
    T u;
    put (&u, 2);
    exch (&u);
    put (&u, 4);
    affi_t (&u);
    return 0;
}

/* prog2.c */
#include "f1.h"
#include "f2.h"
#include "f4.h"
int main () {
    T u;
    put (&u, 2);
    exch (&u); put (&u, 4);
    affi_i(min(get(&u),3));
    return 0;
}
```

2) Écrire un fichier `Makefile` permettant de compiler ces fichiers C et de créer des exécutables, en tenant compte de toutes les dépendances.

3) Créer une cible `all` permettant de déclencher la création de tous les exécutables.

4) Créer une cible `clean` qui supprime tous les fichiers `.o` et tous les exécutables.

II. Script de génération

Écrire le script bash `addmake` admettant les arguments `prog a1.c ... an.c`.
Le script crée un fichier `Makefile` vide s'il n'existe pas déjà, puis rajoute les cibles `prog a1.o ... an.o` et les commandes de compilation, en supposant que ces cibles n'existent pas déjà, et que les fichiers `a1.c ... an.c` sont les fichiers sources nécessaires pour construire l'exécutable `prog`.

TP9 : Makefile

I. Calcul de dépendances

1) Reprendre l'exercice 1 du TD9, taper les fichiers C et le `Makefile` ; tester la compilation, les exécutables produits, la recompilation en cas de modification d'un fichier.

Astuce pour copier-coller le corps des fichiers C depuis l'énoncé en ligne : le visionneur intégré de votre navigateur ne permettant pas de copier individuellement les colonnes, cliquez en haut sur "importer" ou "télécharger", puis ouvrez le document à l'aide du visionneur de documents de votre système. Vous pourrez alors sélectionner individuellement chaque code source.

2) Ajouter une cible `depend` et utiliser la commande `makedepend` pour générer automatiquement les dépendances aux fichiers `.h` ; simplifier le `Makefile` et tester.

3) Ajouter dans le `Makefile` les lignes

```
.c.o :
	gcc -Wall -W -std=c99 -c $*.c
```

puis supprimez toutes les cibles de fichiers `.o` ; tester. Que conclure ?

Arrivé à ce stade, copiez votre `Makefile` sous un autre nom, par exemple `Makefile-v1`, pour ne pas le perdre : en effet dans la question suivante on va *générer* le `Makefile`.

II. Génération de Makefile

1) Reprendre l'exercice 2 du TD9, taper le script bash `addmake` et tester.

2) Modifier le script de telle sorte que seules les cibles qui ne figurent pas déjà dans le `Makefile` y soit rajoutées.

3) Modifier le script de telle sorte que la cible `all` dépende de tous les exécutables que le `Makefile` peut produire.

TD10 : Makefile enrichi

I. Configure

1) Écrire un script `configure` qui génère un fichier `.config` dans le répertoire courant avec la date en commentaire, sauve auparavant l'ancienne version dans `.config.old`, et rétablit l'ancienne version en cas d'interruption (`trap`).

2) Modifier le script pour qu'il définisse dans le fichier `.config` les variables `HAS_CONFIG`, `SHELL`, `CC`, `CFLAGS`, `CPATHS`, `LFLAGS`, `LPATHS`.

3) Écrire un `Makefile` incluant `.config` et utilisant des règles génériques, permettant de compiler complètement un exécutable `prog1` à partir de `f1.c`, `f2.c`, `f3.c` et un exécutable `prog2` à partir de `f2.c` et `f4.c`.

4) Modifier `configure` pour qu'il teste si `make` est une version GNU de la commande (dans ce cas, `make -v` affiche entre autres GNU dans la sortie standard) et définisse alors la variable `HAS_GNUMAKE`.

Modifier le `Makefile` pour qu'il utilise le style approprié de règles génériques en fonction de l'existence de `HAS_GNUMAKE`.

5) Ajouter une cible `clean` permettant d'effacer les `.o` et les exécutables produits.

Ajouter une cible `distclean` dépendant de `clean`, qui supprime `.config` et régénère un `.config` minimal (c'est-à-dire qui ne définit que `SHELL`) avec `configure -z` ; modifier `configure` en conséquence.

6) Dans le `Makefile` faire en sorte que la cible `all` teste si la configuration a été faite avant de provoquer la compilation des exécutables, sinon échoue avec un message d'erreur.

TP10 : Makefile enrichi

I. Configure (suite)

1) Reprendre l'exercice 1 du TD10, taper le script `configure` et le `Makefile` et tester dans l'ordre de l'énoncé. Pour les tests de compilation il peut être utile d'écrire des fichiers `f1.c`, `f2.c`, `f3.c`, `f4.c` tels que ceux-ci :

```
/* f1.h */
#ifndef F1_H
#define F1_H

void affi1 ();

#endif // F1_H
```

```
/* f1.c */
#include <stdio.h>
void affi1 () {
    printf ("1\n");
}
```

```
/* f2.h */
#ifndef F2_H
#define F2_H

void affi2 ();

#endif // F2_H
```

```
/* f2.c */
#include <stdio.h>
void affi2 () {
    printf ("2\n");
}
```

```
/* f3.c */
#include "f1.h"
#include "f2.h"
int main () {
  affi1 (); affi2 ();
  return 0;
}
```

```
/* f4.c */
#include "f2.h"
int main () {
  affi2 ();
  return 0;
}
```

2) Rajouter une cible `depend` qui appelle `gcc -MM` redirigée sur le fichier `.depend`, puis inclure le fichier `.depend`.

3) Modifier le script `configure` pour qu'il demande dans la console un répertoire d'installation, puis définisse la variable `INSTALL_DIR` dans `.config` .

4) Rajouter une cible `install` dans le `Makefile`, de façon à ce que `make install` crée le répertoire d'installation s'il n'existe pas déjà, puis recopie les exécutables dans le répertoire d'installation.

5) Dans le fichier `f4.c` définir une fonction `double puissance(double x, int p)` qui renvoie x^p. Si la macro `USE_MATH` est définie, `f4.c` incluera `math.h` et utilisera `exp(p*log(x))`, sinon utilisera une boucle avec `p` itérations pour calculer x^p.

6) Modifier le script `configure` pour qu'il teste la présence de `libm.so` dans l'un des répertoires `/lib`, `/usr/lib`, `/usr/lib/i386-linux-gnu`, `/usr/lib/x86_64-linux-gnu`, et dans ce cas, rajoute dans `.config` la chaîne `-DUSE_MATH` dans la définition de `CFLAGS`, et `-lm` dans `LFLAGS`.