TP2 — Système d'exploitation

Processus, manipulation de fichiers

Exercice 1 – *Descripteurs de fichier et redirections*

1. Trouver la section correspondant au redirection dans le manuel de BASH, et répondez aux questions suivantes.
   a) Quels opérateurs de redirection permettent d'ouvrir un fichier en lecture ? en écriture ? les deux à la fois ?
   b) À quels modes d'ouverture de fichier correspondent les redirections `<~/.bashrc`, `3<>/tmp/toto` et `>>/tmp/toto.log` ? Quels fichiers sont ouverts ? Quels descripteurs de fichier sont associés ?
   c) À quelles actions correspondent `2>&1`, `3>&-` ?
   d) Que permet la clause de redirection `3>&1 1>&2 2>&3 3>&-` ?
2. Que fait le script suivant ? En particulier, donnez ce qui est affiché sur la sortie standard et sur l'erreur standard.

```
#!/usr/bin/bash
echo "sortie␣de␣$USER" >&1
echo "erreur␣de␣$USER" >&2
echo "écriture␣de␣$USER" >>/tmp/fichier
cat </tmp/fichier >&2
```

3. Créez un tube qui envoie la sortie standard de ce script sur l'entrée standard de la commande `grep` à laquelle vous passerez l'argument `"$USER"`. Que fait `grep "$USER"` ? Précisez-en l'entrée, la sortie et l'erreur standard.
4. Modifiez la ligne de commande de la question précédente, en ajoutant des redirections avant le tube (`|`), afin que l'erreur standard du script soit envoyée à `grep`, et non sa sortie standard – *c.f.* TP1.Exo2.Q5.
5. Pour chacun des cas donnés ci-dessous, remplacez le bloc ⟨`redirections`⟩ de la commande suivante, afin que la sortie contienne [1] la ou les lignes attendues données (faites attention aux permissions) :

```
$ ls -l --time-style=+ /proc/self/fd/ ⟨redirections⟩
```

   a) `lr-x------ 1 [...] 0 -> /tmp/toto`
   b) `lrwx------ 1 [...] 0 -> /tmp/toto`
   c) `l-wx------ 1 [...] 2 -> /tmp/toto`

      `lr-x------ 1 [...] 0 -> /tmp/toto`
      `l-wx------ 1 [...] 2 -> /tmp/toto`
   d) `lrwx------ 1 [...] 3 -> /tmp/toto`
      `lr-x------ 1 [...] 4 -> /tmp/tata`
      `l-wx------ 1 [...] 5 -> /tmp/toto`

1. La sortie contiendra vraisemblablement d'autres lignes, et les ellipses (`[...]`) sur les lignes données contiendront chez vous des informations de propriétés (utilisateur et groupe propriétaire), d'horodatage (date et heure de dernier accès) et de taille (espace mémoire alloué), qui ne nous intéressent pas ici.

Exercice 2 – *Producteur – Consommateur*

Téléchargez les deux scripts `producteur.sh` et `consommateur.sh` et rendez-les exécutables. Le premier script créé un certain nombre (au plus 8) de *tâches* (représentées par des entiers) qu'il donne en sortie standard, à intervalles irréguliers. (Il affiche également quelques messages sur l'erreur standard.) Le second script traite toutes les tâches (représentées par des entiers) lues depuis l'entrée standard (une tâche par ligne), ce qui lui prend un certain temps (variable). Le traitement est représenté par des messages affichés sur la sortie standard. **Vous ne devez pas modifier ces deux scripts.** Vous êtes cependant encouragés à les tester à la main, et à en regarder le code.

1. Écrire un script `prodcons.sh` qui lance 4 producteurs (*i.e.*, 4 processus exécutant `producteur.sh`) et 8 consommateurs (*i.e.*, 8 processus exécutant `consommateur.sh`) en arrière plan, puis attend que tout le monde termine son travail. Les tâches produites par les producteurs doivent toutes être écrite dans **un même** fichier tube, qui sera lu par tous les consommateurs. Ainsi, chaque tâche produite sera traité par un consommateur. Le fichier tube peut être créé préalablement via la commande `mkfifo` (tube nommé). La commande `seq` peut également s'avérer utile.
2. Modifiez votre script pour que tous les messages (pas les tâches envoyées dans le tube) soient affichés sur la sortie standard du script.
3. Modifiez votre script afin que le nombre de producteurs à lancer soit donné en premier argument, et celui de consommateurs à lancer soit donné en second argument.
4. Modifiez votre script pour qu'il accepte zéro, un ou deux arguments. Avec deux arguments, il devra fonctionner comme à la question précédente. Avec un argument (de valeur n), il devra lancer 1 producteur et n consommateurs. Avec zéro argument, il devra lancer 4 producteurs et 8 consommateurs (comme à la **??**). Avec un nombre différent d'arguments, il devra produire un message d'erreur et terminer immédiatement avec le code d'erreur 1.

Exercice 3 – *Interpreteur de commande et capture de signaux*

1. Écrivez un script BASH `interpreter.sh` qui est un interpréteur de commande basique. Le script aura une boucle infinie (bloc `while`), répétant trois étapes :
   1. lire une ligne de l'entrée standard – *c.f.* TP1.Exo1.Q7 ;
   2. évaluer la ligne lue dans un **sous-shell en arrière plan**[2] – *c.f.* TP1.Exo1.Q6 et la commande intégrée `eval` ;
   3. attendre le retour du processus fils – *c.f.* TP1.Exo6.

   Testez[3]. Comment pouvez-vous quitter l'interpréteur ?
2. Modifiez l'interpréteur pour qu'après l'exécution d'une commande, la variable `$status` contienne le code de retour de cette commande. Testez.

2. ce qui correspond, en terme d'appels système, à un `fork` et un `exec` dans le processus fils – *c.f.* TP1.Exo3

3. avec des commandes ne nécessitant pas d'entrée standard car l'envoi en arrière plan la ferme pour le sous-shell

3. La commande `trap` permet de définir le comportement à adopter à la réception d'un signal donné. À l'aide de cette commande, faites en sorte que le signal `SIGINT` (code 2) soit ignoré par notre interpréteur. Ainsi, lorsqu'une commande (*e.g.*, `sleep 180`) est évaluée à l'intérieur de notre interpréteur, l'envoie du signal `SIGINT` (via Ctrl+c) tuera cette commande couramment interprétée mais pas notre interpréteur.

Exercice 4 – *Lecteurs – Écrivains*

Téléchargez le script `lecteur.sh` qui affiche la première et la dernière ligne d'un fichier (dont le chemin est donné en unique argument) séparées par la ligne "...".

1. Écrire un script `ecrivain.sh` qui modifie un fichier en déplaçant sa première ligne en fin de fichier. Le chemin vers le fichier sera donné en unique argument du script. Par exemple, si le fichier `noms.txt` contient :

   ```
   Joséphine Carlier
   Roland Tessier de la Lejeune
   Guillaume Blin
   Laurent Guillon
   ```

   son contenu sera le suivant, après l'appel de `./ecrivain.sh noms.txt` :

   ```
   Roland Tessier de la Lejeune
   Guillaume Blin
   Laurent Guillon
   Joséphine Carlier
   ```

   Pour réaliser cela, vous pourrez utiliser un fichier temporaire (*c.f.* `mktemp`).

2. Les deux scripts `ecrivain.sh` et `lecteur.sh` peuvent-ils être en concurrence ? Y-a-t-il une ressource critique ? Si oui, réalisez un scénario montrant un résultat incohérent (pour cela, vous pouvez tricher, en ralentissant l'exécution des opérations dans les deux scripts – insérez des `sleep .25` adéquates, par exemple).
3. Écrire un script `lecteur-ecrivain.sh` qui lance, en arrière plan, e écrivains et l lecteurs, tous appelés sur un même fichier f où e, l et f sont respectivement donnés en premier, deuxième et troisième argument du script. Une fois lancé, le script attendra que tous (lecteurs et écrivains) finissent. Testez. Constatez-vous des résultats incohérents ?
4. Rétablissez la cohérence à coup sûr, en utilisant l'outil de verrouillage à double phase (*c.f.* cours) `flock` (*c.f.* `man flock` aussi). Avec quelle option doit-il être appelé dans `lecteur.sh` ? Dans `ecrivain.sh` ?
5. À l'aide de la commande `sleep .$RANDOM`, qui force une pause d'une durée aléatoire inférieure à 1 seconde, différez aléatoirement le lancement des processus lecteurs et écrivains dans `lecteur-ecrivain.sh`. Identifiez également chacun par un numéro[4], et faites-en sorte qu'il affiche des messages concernant les opérations qu'ils réalisent sur l'erreur standard. En particulier, on voudra savoir :

4. Vous pouvez ajouter un second argument aux scripts `lecteur.sh` et `ecrivain.sh` afin de leur passer un identifiant déterminé par `lecteur-ecrivain.sh`.

6. quand un verrou est demandé/obtenu/libéré ;
7. quel type de verrou est demandé/obtenu/libéré et par qui.

Exercice 5 – *Signaux*

1. Écrire un script `ilestquelleheure.sh` qui affiche l'heure (à la seconde près) toutes les 5 secondes (dans une boucle infinie), après avoir affiché son **pid**.
2. Exécutez le script et envoyez-lui le signal `SIGUSR1`. Que se passe-t-il ? Tuez le processus s'il est encore en vie.
3. En utilisant la commande `trap`, modifiez votre script pour qu'à la réception du signal `SIGUSR1`, il affiche l'heure (à la seconde près) dans un message de la forme : `"Vous demandez l'heure ? La voici : 09:45:38"`. Le script devra ensuite se remettre en attente. Testez. L'effet est-il immédiat ?
4. La commande `sleep` masque les signaux tant qu'elle s'exécute, d'où un certain délai (d'au plus 5 secondes) qui rend l'envoi du signal tout à fait inintéressant. Fort heureusement, la commande `wait`, elle, ne masque pas les signaux. Modifiez votre script, afin que l'effet de `SIGUSR1` soit immédiat.

5. Interceptez le signal `SIGUSR2` afin que le processus redonne son **pid** avant de continuer (en reprenant le délai à 0)...
6. Interceptez le signal `SIGQUIT` (qui peut être envoyé au processus de premier plan via la combinaison `ctrl+\`) afin qu'un message indique à quelle heure le processus a terminé, avant de terminer.