

ISS - Initiation aux Systèmes d'exploitation et au Shell

LU2IN020

Examen du 14 janvier 2020

Numéro d'anonymat :

Aucun document autorisé pendant l'épreuve

Les téléphones portables, les montres connectées et autres appareils doivent être rangés dans votre sac.
Le barème n'est donné qu'à titre indicatif, pour vous permettre de juger de la difficulté des questions.

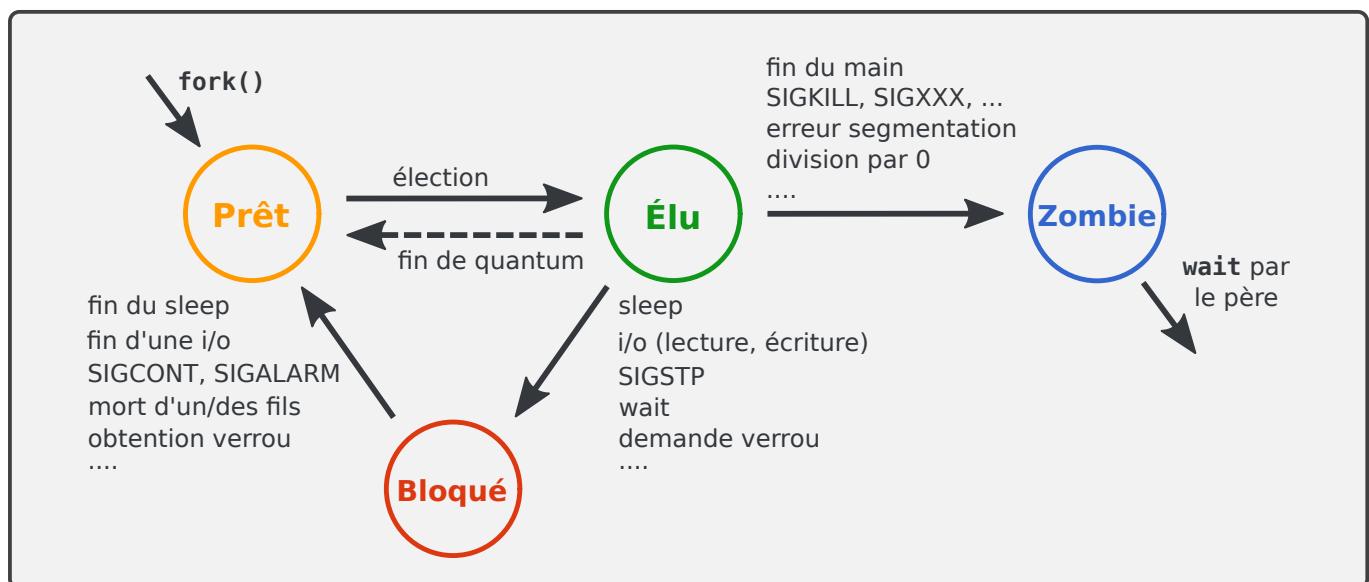
Attention : l'énoncé est imprimé recto-verso sur **9 pages**.

Hypothèse pour l'ensemble de l'examen : Pour simplifier, si les questions n'indiquent pas le contraire, on supposera que tous les exécutables sont bien présents dans le répertoire de l'exercice et que les droits nécessaires à leurs exécutions sont attribués à l'ensemble des utilisateurs.

Exercice 1 : Questions de cours (6 points)

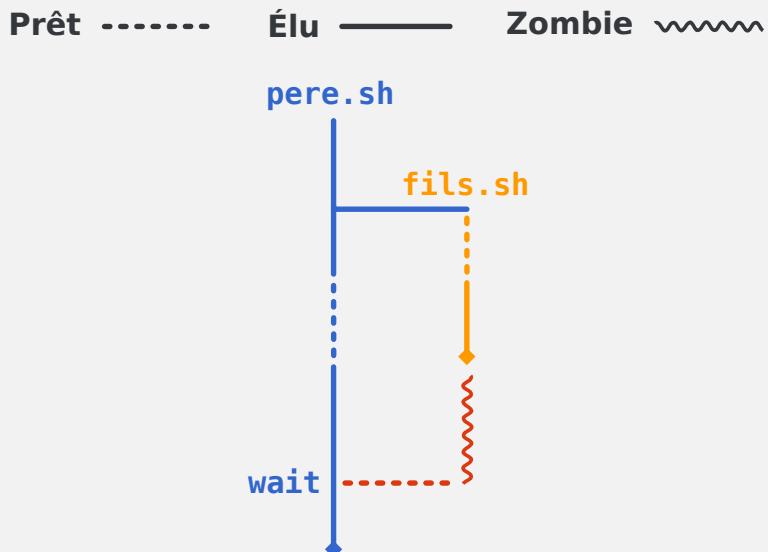
Question 1 – 1,5 point

Dessinez sous forme d'un automate le cycle de vie d'un processus depuis sa création jusqu'à sa mort. Votre schéma devra indiquer l'ensemble des états possibles, ainsi qu'une raison pour chaque changement d'état. Votre schéma devra aussi différencier les systèmes fonctionnant en mode *temps partagé*, de ceux fonctionnant en mode *batch*.



Question 2 – 1 point

Dessinez un chronogramme correspondant à l'exécution *sur un monocœur* du scénario suivant : un processus lance un processus fils en tâche de fond, puis après plusieurs commandes il exécute la commande `wait` alors que son fils est déjà terminé depuis 1 seconde. Votre schéma devra faire apparaître les différents états des processus père et fils tout au long du scénario.



Question 3 – 0,5 point

Quels sont les deux rôles du processus *init* ?

Le processus *init* :

- commence par lancer tous les démons de la machine (sshd, ntpd, cron, ...)
- boucle sur un `wait` pour tuer tous les zombies dont le père originel n'est plus. En effet, *init* adopte tous les processus orphelins.

Question 4 – 1 point

Lorsque l'on fait un `fopen("./mon_fichier.txt",r)` en C ou un `cat "./mon_fichier.txt"` en Bash, qui résout le . et comment ?

C'est le système d'exploitation qui substitue au . le répertoire courant du processus exécutant l'appel système. En effet, il maintient pour chaque processus un répertoire courant qui est hérité du père lors du fork.

Question 5 – 2 points

On considère ici un système dont la variable `rootdir` pointe vers l'inode 8. Le contenu des fichiers correspondant aux inodes 8, 12, 18 et 23 est donné dans ci-dessous.

En modifiant ce schéma, donnez l'état du système de fichier après l'exécution des trois commandes suivantes. Vous pouvez le cas échéant rayer une ligne ou choisir librement un nouveau numéro d'inode.

```
moi@pc / $ mv /dir_2/file_1 /dir_1/dir_1/file_3
moi@pc / $ cp /dir_1/file_1 /dir_2/file_4
moi@pc / $ ln /dir_1/file_2 /file_5
```

inode 8

dir_1	12
dir_2	23
file_1	34
file_2	27

inode 12

dir_1	18
file_1	22
file_2	26

inode 18

file_1	43
file_2	65

inode 23

file_1	52
file_2	37

inode 8

dir_1	12
dir_2	23
file_1	34
file_2	27
file_5	26

inode 12

dir_1	18
file_1	22
file_2	26

inode 18

file_1	43
file_2	65
file_3	52

inode 23

file_1	52
file_2	37
file_4	88

Exercice 2 : The killer (6 points)

Question 1 – 1 point

Quelles lignes de code peut-on ajouter à un script *Bash* pour que le processus qui l'exécute ne s'arrête pas si on lui envoie un SIGTERM ?

```
trap "" SIGTERM
```

Question 2 – 1 point

Peut-on faire de même pour SIGKILL ? Justifiez votre réponse.

Non, car il est impossible de redéfinir la fonction handler du signal SIGKILL.

Question 3 – 1 point

Proposez un nouveau code permettant à un processus de résister à la réception de six signaux SIGTERM mais qui s'arrête à la septième réception.

```
n=0
trap 'n=$((n+1)) ; [ $n -eq 7 ] && exit 0' SIGTERM
```

Question 4 – 1,5 point

On suppose dans la suite de l'exercice que le code de la question 3 ainsi qu'une boucle infinie sont placés dans le fichier `mon_script.sh` du répertoire courant. Il est inutile d'écrire ce script.

Donnez maintenant le code d'un script `test.sh` qui va successivement :

- lancer en tâche de fond un script `mon_script.sh`
- envoyer `n` fois le signal `SIGTERM` à cette tâche, où `n` est la valeur passée en paramètre au script `test.sh` s'il est appelé avec un paramètre, sinon on utilisera par défaut la valeur 7.
- se terminer en affichant "`fin du test`" après que `mon_script.sh` se soit arrêté.

```
#!/usr/bin/env bash

num=7
[ $# -gt 0 ] && num=$1

mon_script.sh &
pid=$!

for i in $( seq 1 $num ) ; do
    kill -SIGTERM $pid
done

wait
echo "fin du test"
```

Question 5 – 0,5 point

Pour quelle valeur du paramètre peut-on être certain que le script `test.sh` termine ? Pourquoi ?

On ne peut jamais être certain que le script se termine, car l'envoi de `x` signaux à un processus peut conduire à l'exécution d'un seul handler du signal. C'est le cas si tous les signaux sont envoyés avant que le processus destinataire n'exécute un handler.

Notons que l'ajout d'un `sleep`, qui forcerait la commutation, n'est pas suffisant. En effet, le script `test.sh` peut être réélu avant que `mon_script.sh` ne puisse exécuter son handler.

Question 6 – 1 point

Proposez une nouvelle version de `test.sh` qui assure réellement l'arrêt du script `mon_script.sh` lancé en tâche de fond. Cette nouvelle version ne prendra pas de paramètre et est libre d'envoyer tous les signaux `SIGTERM` nécessaires (vous n'avez pas ici le droit d'utiliser d'autres types de signaux). Vous pouvez utiliser la commande `ps <pid>` qui retourne 0 s'il existe un processus avec le pid passé en paramètre et 1 dans le cas contraire.

```
#!/usr/bin/env bash

mon_script.sh &
pid=$!

while ps $pid > /dev/null ; do
    kill -SIGTERM $pid
done

wait
echo "fin du test"
```

Exercice 3 : Multigrep (4 points)

Dans cet exercice, on veut implémenter un *multigrep* qui permet de paralléliser un grep. Pour simplifier, on se contentera ici de la recherche des lignes contenant une chaîne de caractères dans un fichier. Cette commande s'utilisera de la façon suivante :

```
moi@pc /home/moi $ multigrep chaîne fichier nb_processus_grep
```

Question 1 – 1,5 point

Pour commencer, implémenter un script *my_split* qui prend en paramètre un nom de fichier, ainsi qu'un nombre *n* et qui découpe ce fichier en *n* petits fichiers ayant un même nombre de lignes. Le nom de ces fichiers correspond au nom du fichier découpé suffixé par un numéro (nom.1, nom.2, nom.3). Vous n'avez pas ici le droit d'utiliser la commande *split*.

Pour simplifier, on supposera que le nombre de lignes dans le fichier à découper est un multiple de *n* et vous n'êtes pas obligés de respecter l'ordre des lignes dans les fichiers.

Solution utilisant la commande **read** (qui ne préserve pas l'ordre) :

```
#!/usr/bin/env bash

while read ligne ; do
    echo $ligne >> $1.1
    for i in $(seq 2 $2) ; do
        read ligne
        echo $ligne >> $1.$i
    done
done < $1
```

Solution utilisant la commande **head** :

```
#!/usr/bin/env bash

fichier=$1
size_tail=$(wc -l < $fichier)
size_head=$((size_tail / $2))

for i in $( seq 1 $2 ) ; do
    tail -n $size_tail $fichier | head -n $size_head >> $fichier.$i
    size_tail=$((size_tail-size_head))
done
```

Solution utilisant la commande **sed** :

```

#!/usr/bin/env bash

fichier=$1
size=$(( $(wc -l < $fichier) / $2 - 1))

for i in $( seq 1 $2 ) ; do
    debut=$((fin+1))
    fin=$((debut+size))
    sed -n "${debut},${fin}p" $fichier >> $fichier.$i
done

```

Question 2 – 0,5 point

Sachant que votre script se trouve dans le répertoire `/home/moi/mesScripts`, que doit-on ajouter et dans quel fichier pour qu'on puisse l'utiliser comme indiqué en début dexercice quel que soit le répertoire courant ?

Il faut ajouter ce répertoire à la variable `$PATH` en ajoutant dans le fichier `.bashrc` la ligne `PATH=$PATH:/home/moi/mesScripts`.

Question 3 – 1,5 point

À l'aide de votre commande `my_split` (que l'on supposera ici correcte), implémentez-en *Bash* la commande `multigrep` désirée. Votre code devra découper le fichier en `n` petits fichiers sur lesquels vous lancerez autant de grep en concurrence. Votre commande s'arrêtera lorsque tous les grep seront terminés.

```

#!/bin/bash

string=$1
file=$2
num=$3

my_split $file $num

for i in $( seq 1 $num ) ; do
    grep $string $file.$num &
done

wait

```

Question 4 – 0,5 point

Si le principe de la parallélisation permet d'utiliser au mieux les architectures multicoeurs, il peut aussi améliorer les performances sur un monocœur. Pourquoi ?

Le lancement en parallèle de plusieurs processus peut permettre de masquer le temps de blocage dû aux entrées sorties : lorsqu'un processus attend ses données, un autre s'exécute.

Exercice 4 : Le transporteur (5 points)

Dans cet exercice on s'intéresse au traitement d'une liste de transfert de mails `forward.csv`. Sur cette liste au format *csv*, chaque ligne est composée de deux adresses mail séparées par une virgule. Les adresses mail considérées sont formées d'un login (composé uniquement de lettres), suivi d'une arobase, puis d'un nom de domaine (deux mots séparés par un point). Dans la suite de cet exercice, on nommera *adresse source* la première adresse d'une ligne et *adresse de destination* la deuxième.

Voici un exemple de fichier (attention vos scripts devront marcher pour tout fichier respectant le format précédemment décrit) :

```
jojo@hotmail.com,jonathan@yahoo.com
winstonsmith@gmail.com,winstonsmith@protonmail.com
johnny@gmail.com,jonathan@yahoo.com
cirelli@engie.fr,cirelli@blackrock.com
...
...
```

Dans chacune des questions, on va vous demander de donner une commande permettant de résoudre un problème. Pour chaque réponse, vous êtes autorisé à utiliser l'ensemble des commandes *Bash* étudiées et, le cas échéant, à les composer entre elles avec des tubes.

Question 1 – 1 point

Donnez une commande qui permette d'afficher le contenu du fichier `forward.csv` en ayant transformé tous les "@" en "at".

```
moi@pc /home/moi $ sed -E 's/@/at/g' forward.csv
```

Question 2 – 1 point

Donnez une commande qui affiche tous les logins des adresses sources du domaine *free.fr* que l'on transfère à une adresse du domaine *gmail.com*.

```
moi@pc /home/moi $ grep -E '@free.fr,.*@gmail.com$' forward.csv | cut -d @ -f 1
moi@pc /home/moi $ sed -E '@free.fr,.*@gmail.com$/!d;s/@.*//' forward.csv
```

Question 3 – 1 point

Donnez une commande qui affiche le nombre d'adresses de destination différentes dans le fichier.

```
moi@pc /home/moi $ cut -d , -f2 forward.csv | sort | uniq | wc -l
moi@pc /home/moi $ cut -d , -f2 forward.csv | sort -u | wc -l
moi@pc /home/moi $ sed 's/.*///' forward.csv | sort -u | wc -l
```

Question 4 – 1 point

Donnez une commande qui affiche la liste des adresses destinations dont les adresses sources ont des logins formés d'une répétition de deux lettres (*e.g.*, jojo).



```
moi@pc ~ % grep -E "^(..)\1@" forward.csv | cut -d , -f 2
moi@pc ~ % sed -E 's/(..)\1@/!d;s/.*/' forward.csv
```

Question 5 – 1 point

Donnez une commande qui génère un fichier `clean_forward.csv` correspondant contenant les mêmes informations que la liste `forward.csv` mais sans aucune majuscule. Les lettres majuscules présentes dans le fichier original devront avoir été remplacées par leur équivalent en minuscule.

```
moi@pc ~ % tr A-Z a-z < forward.csv > clean_forward.csv
moi@pc ~ % sed y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
forward.csv > clean_forward.csv
moi@pc ~ % sed 's/.*/\L&/' forward.csv > clean_forward.csv
```

Exercice 5 : Synchronisation (1,5 point bonus)

Question 1 – 0,5 point

Nous avons étudié cette année deux types d'utilisation des mécanismes de synchronisation. Quels sont-ils ? Vous n'avez pas ici à les définir, seul le nom du type ou de la famille de problèmes suffit.

Les deux types d'utilisation étudiés cette année sont :

- le verrouillage de ressources (exclusion mutuelle)
- les barrières

Question 2 – 1 point

On considère les scripts `A.sh` et `B.sh` donnés ci-dessous. Ces scripts utilisent la commande `iss_synchro` fournie en TP et l'on considère que le `verrou` a déjà été créé.

`A.sh`

```
#!/bin/bash

echo A1
iss_synchro lock verrou
echo A2
echo A3
iss_synchro unlock verrou
```

`B.sh`

```
#!/bin/bash

echo B1
iss_synchro lock verrou
echo B2
iss_synchro unlock verrou
echo B3
```

On suppose qu'on lance dans une console ces deux scripts en concurrence. Indiquez pour chacun des affichages suivants, s'ils sont possibles ou impossibles (dans ce dernier cas, vous indiquez pourquoi c'est impossible) :

- Ordre d'affichage 1 : A1 A2 B1 B2 A3 B3
- Ordre d'affichage 2 : A1 A2 B1 A3 B2 B3
- Ordre d'affichage 3 : A1 B1 B2 A2 B3 A3



Les affichages 2 et 3 sont possibles, mais ce n'est pas le cas de l'affichage 1. En effet, entre les affichages A2 et A3, on sait que `A.sh` possède le verrou. Il n'est donc pas possible que B2 s'intercale entre ces deux affichages, car `B.sh` a besoin du verrou pour le faire.

