3R-IN3 – Systèmes d’exploitation

TP 2

Xavier Hilaire, Clément Boin
x.hilaire@esiee.fr, clement.boin@gmail.com

19 mars 2025

1 Réponse en temps limité

La commande interne `read` du shell permet de lire une ligne à partir du terminal, mais attend indéfiniment que l'utilisateur veuille bien en saisir une. Dans cet exercice, on cherche à écrire un script qui attend la réponse de l'utilisateur en un temps limité : s'il ne saisit rien au-delà d'une limite de temps, le script doit supposer une réponse par défaut, et poursuivre son exécution.

1.1 Script `question.sh`, première version

Écrivez un premier script `question.sh` qui fera les choses suivantes :

- Il affiche son PID sur sa sortie d'erreur standard
- Il affiche son unique argument, supposé être une question
- Il lit la réponse de l'utilisateur, qu'on supposera de type oui/non
- Si l'utilisateur a répondu `0` ou `o`, le script doit terminer avec le code de sortie `0`, sinon avec `1`.

Exemple de résultat attendu :

```
[xavier@localhost TD]$ ./question.sh "Aimez-vous la tarte au concombre (O/N) ? "  
PID = 14473  
Aimez-vous la tarte au concombre (O/N) ? o  
[xavier@localhost TD]$ echo $?  
0  
[xavier@localhost TD]$ ./question.sh "Aimez-vous la tarte au concombre (O/N) ? "  
PID = 14482  
Aimez-vous la tarte au concombre (O/N) ? 0  
[xavier@localhost TD]$ echo $?  
0  
[xavier@localhost TD]$ ./question.sh "Aimez-vous la tarte au concombre (O/N) ? "  
PID = 14493  
Aimez-vous la tarte au concombre (O/N) ? pas du tout  
[xavier@localhost TD]$ echo $?  
1  
[xavier@localhost TD]$
```

Sol.:

```
#!/bin/sh  
#
```

```

echo "PID = $$" >& 2
echo -n "$1"
read reponse
{
    test "$reponse" = o || test "$reponse" = 0
} && exit 0
exit 1

```

1.2 Script question.sh, deuxième version

- Modifiez le script précédent pour garantir qu’il termine au bout de 5 secondes, même si l’utilisateur ne saisit rien. Vous pourrez utiliser un autre script ou processus, et les commandes `kill` et `sleep`, vues en cours, pour ce faire. Exemple de résultat attendu :

```

[xavier@localhost TD]$ date ; ./question.sh "Aimez-vous la tarte au concombre (O/N) ?
sam. 20 mai 2023 10:33:28 CEST
PID = 14975
Aimez-vous la tarte au concombre (O/N) ?
[xavier@localhost TD]$ date
sam. 20 mai 2023 10:33:33 CEST
[xavier@localhost TD]$

```

Sol.: On peut commencer par créer un nouveau script `tueur.sh` contenant les lignes suivantes :

```

#!/bin/sh

sleep 5
kill -INT $1

```

Ce script “tue” le processus dont il reçoit le PID en argument après avoir attendu 5s. L’idée est alors de lancer ce script en arrière-plan dans `question.sh`, avec le PID de `question.sh` lui-même ; ce qui revient à dire que `question.sh` demande à l’un de ses enfants de le tuer au bout de 5s s’il est encore en vie (ce qui arrivera si l’utilisateur n’a rien fait) :

```

#!/bin/sh

echo "PID = $$" >& 2
tueur.sh $$ &
echo -n "$1"
read reponse
{
    test "$reponse" = o || test "$reponse" = 0
} && exit 0
exit 1

```

On peut aussi lancer directement les appels à `sleep` et `kill` dans un sous-shell, ce qui évite de créer un troisième script :

```

#!/bin/sh

```

```

echo "PID = $$" >& 2
(sleep 5 ; kill -INT $$) &
echo -n "$1"
read reponse
{
    test "$reponse" = o || test "$reponse" = 0
} && exit 0
exit 1

```

- Examinez le code de sortie retourné par le shell dans ce cas de figure.

Sol.:

```

[xavier@localhost TD]$ ./question.sh "Aimez-vous la tarte au concombre (O/N) ? "
PID = 15256
Aimez-vous la tarte au concombre (O/N) ?
[xavier@localhost TD]$ echo $?
130
[xavier@localhost TD]$

```

Le code 130 est celui retourné par le shell en cas d'interruption par SIGINT. Cet entier peut varier d'un signal à l'autre, et d'un système à l'autre.

- Votre script est-il sans danger dans le cas où l'utilisateur saisit quelque chose dans les temps ? S'il y a un risque, peut-on le corriger, et comment ?

Sol.: Il subsiste un danger dans ce cas précis, car la terminaison de `question.sh` dans les temps n'entraîne pas celle de `tueur.sh` ou du sous-shell. On voit d'ailleurs s'afficher un message comparable au suivant dans ce cas de figure :

```

[xavier@localhost TD]$ ./question.sh: ligne 5 : kill: (15347) - Aucun processus de ce

```

Le shell enfant continue d'exister même si son père est décédé, et il cherche à tuer un processus qui n'existe plus. Le danger est alors que si ce numéro de PID est réutilisé pour identifier un autre processus, alors le shell fils le tuera quand bien même il n'a rien à voir avec le problème traité. Pour limiter ce risque, une solution consisterait à ce que `question.sh` tue son sous-shell si l'utilisateur répond dans les temps ; autrement dit, à récupérer le PID du sous-shell immédiatement après son lancement, puis faire un appel à `kill` dès que le `read` est passé, comme dans le code suivant :

```

#!/bin/sh
#

echo "PID = $$" >& 2
(sleep 5; kill -INT $$) &
fils=$!
echo -n "$1"
read reponse
kill $fils
{
    test "$reponse" = o || test "$reponse" = 0
} && exit 0
exit 1

```

Cette solution n'est toutefois pas parfaite : comme nous le verrons l'année prochaine

durant l'unité de programmation système, il n'y a aucune garantie que lorsqu'une fois la commande `kill $fils` terminée, le fils soit effectivement mort, car les signaux sont traités de manière asynchrone et différée dans le temps.

1.3 Script principal

Finalement, écrivez et testez un script `principal.sh` qui appellera `question.sh`, et affichera l'un des messages suivants selon le code de retour qu'il aura reçu :

0	Je savais que vous répondriez oui
1	Je savais que vous répondriez non
autre	Vous êtes lent !

Sol.:

```
#!/bin/sh
#

./question.sh "Aimez-vous la tarte au concombre (O/N) ? "
case $? in
  0) echo "Je savais que vous répondriez oui !"
    ;;
  1) echo "Je savais que vous répondriez non !"
    ;;
  *) echo "Vous êtes lent !"
    ;;
esac
```

2 Scanneur ICMP

La commande `ping -c 1 -w 1 machine` permet de tester la réponse ICMP d'une machine spécifiée à l'aide d'un seul paquet ICMP, et en n'attendant pas plus d'une seconde. Examinez son code de sortie sur une machine ayant toutes les chances de répondre (127.0.0.1 par exemple), et peu de chances de répondre (10.0.0.15 par exemple).

Sachant cela, écrire un script `scan.sh [-p] [-o fichier] adresse/nn` qui fait les choses suivantes :

- L'ensemble `adresse/nn` est une adresse IPv4 de réseau en notation condensée CIDR. On supposera que `nn` peut valoir 32, 24, ou 16, et on ne traitera pas les autres cas – vous pourrez vous contenter d'un message « cas `nn` non traité »
- Le script doit tester la réponse ICMP de l'ensemble des machines adressables par l'adresse CIDR fournie. Vous devrez a minima faire ces tests séquentiellement, mais vous pourrez faire mieux en paralléliser les tests par lots de 256. En d'autres termes, vous pouvez lancer un lot de 256 tests parallèles, puis lancer le lot suivant (si c'est nécessaire) une fois tous les tests du lot courant terminés. Conseil : écrivez et validez d'abord la version séquentielle, la version parallèle s'en déduisant presque immédiatement.
- Lorsqu'une machine répond à un ping, le script doit afficher un message de la forme « La machine `xx.yy.zz.tt` » a répondu a ping » et rien d'autre.
- Lorsqu'une machine ne répond pas à un ping, le script doit afficher un message de la forme « La machine `xx.yy.zz.tt` » n'a pas répondu a ping » et rien d'autre, sauf lorsque l'option `-p` a été utilisée
- Si l'option `-o fichier` a été utilisée, alors le script doit écrire non pas sur la sortie standard, mais dans le `fichier` spécifié plutôt qu'à l'écran.

On ne doit voir s'afficher aucun des messages écrits par ping à l'écran. Conseils :

- Commencez par écrire un autre script, ou mieux, une fonction, qui teste une adresse IP particulière en affichant ce qu'il faut et là où il faut
- Appelez ce script ou cette fonction d'abord séquentiellement, puis parallèlement
- Traitez les cas 32, 24 et 16 dans cet ordre
- Le domaine esiee.fr est de classe B, et toutes les machines sont en 147.215.0.0/16. Mais très peu répondent au ping. Essayez principalement de tester les réponses sur le réseau en /24 sur lequel se trouve la vôtre, il est peu probable que les tests au-delà soient représentatifs.

Sol.:

Version séquentielle :

```
#!/bin/sh

# Scanneur ICMP, version séquentielle

# Mémoriser les options
if test "$1" = "-p"; then
    negopt=yes
    shift
fi
if test "$1" = "-o"; then
    > "$2"
    sortie=">> $2"
    shift 2
fi

# Suffixe et préfixe
suf=${1#*/}
pref=${1%/*}

# Fonction pour tester une IP particulière
function xtester() {
    if ping -c 1 -w 1 "$1" >/dev/null 2>&1; then
        eval echo "La machine $1 a répondu à ping" $sortie
    else
        if test \! "$negopt"; then
            eval echo "La machine $1 n\'a pas répondu à ping" $sortie
        fi
    fi
}

case $suf in
    32) xtester $pref
        fi
        ;;
    24) pref=${pref%.*} # raccourcir le préfixe d'une décimale
        i=0
        while test $i -le 255; do
            xtester ${pref}.$i
            i=$((i+1))
        done
esac
```

```

done
;;
16) pref=${pref%.*.*} # raccourcir le préfixe de deux décimales
j=0
while test $j -le 255; do
    i=0
    while test $i -le 255; do
        xtester ${pref}.${j}.${i}
        i=$((i+1))
    done
    j=$((j+1))
done
;;

*) echo "Cas non géré"
exit 1
;;
esac

```

Version parallèle :

```
#!/bin/sh
```

```
# Scanneur ICMP, version parallèle
```

```
# Mémoriser les options
```

```
if test "$1" = "-p"; then
```

```
    negopt=yes
```

```
    shift
```

```
fi
```

```
if test "$1" = "-o"; then
```

```
    > "$2"
```

```
    sortie=">> $2"
```

```
    shift 2
```

```
fi
```

```
# Suffixe et préfixe
```

```
suf=${1#*/}
```

```
pref=${1%/*}
```

```
# Fonction pour tester une IP particulière
```

```
function xtester() {
```

```
    if ping -c 1 -w 1 "$1" >/dev/null 2>&1; then
```

```
        eval echo "La machine $1 a répondu à ping" $sortie
```

```
    else
```

```
        if test \! "$negopt"; then
```

```
            eval echo "La machine $1 n\'a pas répondu à ping" $sortie
```

```
        fi
```

```
    fi
```

```
}
```

```
case $suf in
```

```

32) xtester $pref
    ;;
24) pref=${pref%.*} # raccourcir le préfixe d'une décimale
    i=0
    while test $i -le 255; do
        xtester ${pref}.$i &
        i=$((i+1))
    done
    wait
    ;;
16) pref=${pref%.*.*} # raccourcir le préfixe de deux décimales
    j=0
    while test $j -le 255; do
        i=0
        while test $i -le 255; do
            xtester ${pref}.$j.$i &
            i=$((i+1))
        done
        wait
        j=$((j+1))
    done
    ;;

*) echo "Cas non géré"
   exit 1
   ;;
esac

```

3 Processus

Source : C. Boin

1. Lancez votre terminal
2. Avec l'éditeur « vim » ou « nano », saisir ce programme, nommé « pg_erreur », dans le répertoire courant.

```

#!/bin/bash

while (true)
do
echo Bonjour
done

```
3. Enregistrez-le.
4. Rendez-le fichier « pg_erreur » exécutable.
5. Exécutez-le.
6. Que se passe-t-il ?
7. Arrêtez le programme par Ctrl-C.
8. Relancez le programme mais en faisant suivre son nom par un « & ».
9. Arrêtez le programme par Ctrl-C. Que constatez-vous ?
10. Pourquoi ?

11. Allez sur TTY2 et connectez-vous (Ctrl Droit + F2)
12. La commande « ps » liste tous les processus actifs de la session utilisateur. Quelle commande doit-t-on taper pour voir tous les processus en cours ?
13. Recherchez le script que vous venez d'écrire. Comment le reconnaissez-vous ?
14. Quel est son identifiant (PID) ?
15. Quel est l'identifiant de son père ?
16. Lancer la commande « top ». Où est situé le programme « pg-erreur » ?
17. Quel est le processus qui consomme le plus de temps processeur ?
18. Quittez « top » en appuyant sur « q ».
19. Tuez le processus de « pg-erreur » à l'aide de la commande « kill » (ou avec « top »).
20. Vérifiez avec « top » que tout est rentré dans l'ordre. Comment le voyez-vous ?
21. Relancez à nouveau le script précédent, et interrompez-le avec la commande Ctrl-Z, que se passe-t'il ?
22. Tapez la commande « fg », que se passe-t'il ?
23. Tapez à nouveau Ctrl-Z, puis tapez la commande « bg », que se passe-t'il ? Et comment se sortir de cette situation ?
24. Expliquer brièvement le fonctionnement des commandes « nice » et « nohup »