19. Redirection d'E/S (entrées/sorties)

Trois différents « fichiers » sont toujours ouverts par défaut, `stdin` (le clavier), `stdout` (l'écran) et `stderr` (la sortie des messages d'erreur vers l'écran). Ceux-ci, et n'importe quel autre fichier ouvert, peuvent être redirigés. La redirection signifie simplement la capture de la sortie d'un fichier, d'une commande, d'un programme, d'un script, voire même d'un bloc de code dans un script (voir l'Exemple 3.1, « Blocs de code et redirection d'entrées/sorties » et l'Exemple 3.2, « Sauver la sortie d'un bloc de code dans un fichier ») et le renvoi du flux comme entrée d'un autre fichier, commande, programme ou script.

Chaque fichier ouvert se voit affecté un descripteur de fichier. [82] Les descripteurs de fichier pour `stdin`, `stdout` et `stderr` sont, respectivement, 0, 1 et 2. Pour ouvrir d'autres fichiers, il reste les descripteurs 3 à 9. Il est quelque fois utile d'affecter un de ces descripteurs supplémentaires de fichiers pour `stdin`, `stdout` ou `stderr` comme lien dupliqué temporaire. [83] Ceci simplifie le retour à la normale après une redirection complexe et un remaniement (voir l'Exemple 19.1, « Rediriger `stdin` en utilisant *exec* »).

```
SORTIE_COMMANDE >
   # Redirige la sortie vers un fichier.
   # Crée le fichier s'il n'est pas présent, sinon il l'écrase.

   ls -lR > repertoire.liste
   # Crée un fichier contenant la liste des fichiers du répertoire.

: > nom_fichier
   # Le > vide le fichier "nom_fichier".
   # Si le fichier n'est pas présent, crée un fichier vide (même effet que
   # 'touch').
   # Le : sert en tant que contenant, ne produisant aucune sortie.

> nom_fichier
   # Le > vide le fichier "nom_fichier".
   # Si le fichier n'est pas présent, crée un fichier vide (même effet que
   # 'touch').
   # (Même résultat que ": >", ci-dessus, mais ceci ne fonctionne pas avec
   # certains shells.)

SORTIE_COMMANDE >>
   # Redirige stdout vers un fichier.
   # Crée le fichier s'il n'est pas présent, sinon il lui ajoute le flux.


   # Commandes de redirection sur une seule ligne (affecte seulement la ligne
   # sur laquelle ils sont):
   # --------------------------------------------------------------------

1>nom_fichier
   # Redirige stdout vers le fichier "nom_fichier".
1>>nom_fichier
   # Redirige et ajoute stdout au fichier "nom_fichier".
2>nom_fichier
   # Redirige stderr vers le fichier "nom_fichier".
2>>nom_fichier
   # Redirige et ajoute stderr au fichier "nom_fichier".
&>nom_fichier
   # Redirige à la fois stdout et stderr vers le fichier "nom_fichier".
   #
   #  Notez que   &>>fichier
   #+ -- tente de rediriger et d'*ajouter*
```

```
   #+ stdout et stderr au fichier "fichier" --
   #+ échoue avec un message d'erreur,
   #+ syntax error near unexpected token '>'.

M>N
  # "M" est un descripteur de fichier, par défaut 1 s'il n'est pas précisé.
  # "N" est un nom de fichier.
  # Le descripteur de fichier "M" est redirigé vers le fichier "N".
M>&N
  # "M" est un descripteur de fichier, par défaut 1 s'il n'est pas précisé.
  # "N" est un autre descripteur de fichier.

   #============================================================================

   # Rediriger stdout, une ligne à la fois.
   FICHIERLOG=script.log

   echo "Cette instruction est envoyée au fichier de traces, \"$FICHIERLOG\"." 1>$FICHIERLOG
   echo "Cette instruction est ajoutée à \"$FICHIERLOG\"." 1>>$FICHIERLOG
   echo "Cette instruction est aussi ajoutée à \"$FICHIERLOG\"." 1>>$FICHIERLOG
   echo "Cette instruction est envoyé sur stdout et n'apparaitra pas dans \"$FICHIERLOG\"."
   # Ces commandes de redirection sont "réinitialisées" automatiquement après chaque ligne.



   # Rediriger stderr, une ligne à la fois.
   FICHIERERREURS=script.erreurs

   mauvaise_commande1 2>$FICHIERERREURS     #  Message d'erreur envoyé vers $FICHIERERREURS.
   mauvaise_commande2 2>>$FICHIERERREURS #  Message d'erreur ajouté à $FICHIERERREURS.
   mauvaise_commande3                          #  Message d'erreur envoyé sur stderr,
                                               #+ et n'apparaissant pas dans $FICHIERERREURS.
   #  Ces commandes de redirection sont aussi "réinitialisées" automatiquement
   #+ après chaque ligne.
   #============================================================================
```

```
2>&1 \
   # Redirige stderr vers stdout.
   # Les messages d'erreur sont envoyés à la même place que la sortie standard.

i>&j
   # Redirige le descripteur de fichier i vers j.
   # Toute sortie vers le fichier pointé par i est envoyée au fichier pointé par j.

>&j
   # Redirige, par défaut, le descripteur de fichier 1 (stdout) vers j.
   # Toutes les sorties vers stdout sont envoyées vers le fichier pointé par j.
```

```
0< NOM_FICHIER
 < NOM_FICHIER
   # Accepte l'entrée à partir d'un fichier.
   # Commande compagnon de « > », et souvent utilisée en combinaison avec elle.
   #
   # grep mot_recherché <nom_fichier


[j]<>nom_fichier
   #  Ouvre le fichier "nom_fichier" pour lire et écrire, et affecter le descripteur
```

```
      #+ de fichier "j" à celui-ci.
      # Si "nom_fichier" n'existe pas, le créer.
      # Si le descripteur de fichier "j" n'est pas spécifié, le défaut est fd 0, stdin.
      #
      # Une application de ceci est d'écrire à une place spécifiée dans un fichier.
      echo 1234567890 > Fichier    # Écrire une chaîne dans "Fichier".
      exec 3<> Fichier             # Ouvrir "Fichier" et lui affecter le fd 3.
      read -n 4 <&3                # Lire seulement quatre caractères.
      echo -n . >&3                # Écrire un point décimal à cet endroit.
      exec 3>&-                   # Fermer fd 3.
      cat Fichier                  # ==> 1234.67890
      # Accès au hasard, par golly.



   |
      # Tube.
      # outil de chaînage de processus et de commande à but général.
      # Similaire à « > », mais plus général dans l'effet.
      # Utile pour chaîner des commandes, scripts, fichiers et programmes.
      cat *.txt | sort | uniq > fichier-resultat
      # Trie la sortie de tous les fichiers .txt et supprime les lignes
      # dupliquées, pour finalement enregistrer les résultats dans
      # « fichier-résultat ».
```

Plusieurs instances de redirection d'entrées et de sorties et/ou de tubes peuvent être combinées en une seule ligne de commande.

```
commande < fichier-entrée > fichier-sortie

commande1 | commande2 | commande3 > fichier-sortie
```

Voir l'Exemple 15.31, « Déballer une archive *rpm* » et l'Exemple A.15, « *fifo*: Faire des sauvegardes journalières, en utilisant des tubes nommés ».

Plusieurs flux de sortie peuvent être redirigés vers un fichier.

```
ls -yz >> commande.log 2>&1
# La capture résulte des options illégales "yz" de "ls" dans le fichier
# "commande.log".
# Parce que stderr est redirigé vers le fichier, aucun message d'erreur ne sera
# visible.

# Néanmoins, notez que ce qui suit ne donne *pas* le même résultat.
ls -yz 2>&1 >> command.log
# Affiche un message d'erreur et n'écrit pas dans le fichier.

#  Si vous redirigez à la fois stdout et stderr, l'ordre des commandes fait une
#+ différence.
```

Fermer les descripteurs de fichiers

- **n<&-**
  Ferme le descripteur de fichier *n*.
- **0<&-, <&-**
  Ferme `stdin`.
- **n>&-**

Ferme le descripteur de fichiers de sortie *n*.

- **1>&-, >&-**
  Ferme `stdout`.

Les processus fils héritent des descripteurs de fichiers ouverts. C'est pourquoi les tubes fonctionnent. Pour empêcher l'héritage d'un fd, fermez-le.

```
# Rediriger seulement stderr vers un tube.

exec 3>&1                             # Sauvegarde la valeur "actuelle" de stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-   # Ferme fd 3 pour 'grep' (mais pas pour 'ls').
#              ^^^^   ^^^^
exec 3>&-                             # Maintenant, fermez-le pour le reste du script.

# Merco, S.C.
```

Pour une introduction plus détaillée de la redirection d'E/S, voir l'Annexe E, Une introduction détaillée sur les redirections d'entrées/sorties.

19.1. Utiliser exec

Une commande **exec <nomfichier** redirige `stdin` vers un fichier. À partir de là, `stdin` vient de ce fichier plutôt que de sa source habituelle (généralement un clavier). Ceci fournit une méthode pour lire un fichier ligne par ligne et donc d'analyser chaque ligne de l'entrée en utilisant sed et/ou awk.

Exemple 19.1. Rediriger `stdin` en utilisant *exec*

```
#!/bin/bash
# Rediriger stdin en utilisant 'exec'.


exec 6<&0          # Lie le descripteur de fichier #6 avec stdin.
                   # Sauvegarde stdin.

exec < fichier-donnees   # stdin remplacé par le fichier "fichier-donnees"

read a1            # Lit la première ligne du fichier "fichier-donnees".
read a2            # Lit la deuxième ligne du fichier "fichier-donnees".

echo
echo "Les lignes suivantes lisent le fichier."
echo "---------------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Maintenant, restaure stdin à partir de fd #6, où il a été sauvegardé,
#+ et ferme fd #6 ( 6<&- ) afin qu'il soit libre pour d'autres processus.
#
# <&6 6<&-    fonctionne aussi.

echo -n "Entrez des données "
read b1  #  Maintenant les fonctions lisent ("read") comme d'ordinaire,
         #+ c'est-à-dire à partir de stdin.
echo "Entrée lue à partir de stdin."
```

```
echo "-----------------------------"
echo "b1 = $b1"

echo

exit 0
```

De la même façon, une commande **exec >nomfichier** redirige `stdout` vers un fichier désigné. Ceci envoie toutes les sorties des commandes qui devraient normalement aller sur `stdout` vers ce fichier.

Important

exec N > nom_fichier affecte le script entier ou le ***shell courant***. La redirection vers le PID du script ou du shell à partir de maintenant à changer. Néanmoins...

N > nom_fichier affecte seulement les processus qui vont être créés, pas le script entier ou le shell.

Merci à Ahmed Darwish de nous l'avoir précisé.

Exemple 19.2. Rediriger `stdout` en utilisant *exec*

```
#!/bin/bash
# reassign-stdout.sh

FICHIERTRACES=fichiertraces.txt

exec 6>&1           # Lie le descripteur de fichier #6 avec stdout.
                    # Sauvegarde stdout.

exec > $FICHIERTRACES     # stdout remplacé par le fichier "fichiertraces.txt".

# ----------------------------------------------------------- #
#  Toute sortie des commandes de ce bloc sera envoyée dans le fichier
#+ $FICHIERTRACES.

echo -n "Fichier traces: "
date
echo "-----------------------------------"
echo

echo "Sortie de la commande \"ls -al\""
echo
ls -al
echo; echo
echo "Sortie de la commande \"df\""
echo
df

# ----------------------------------------------------------- #

exec 1>&6 6>&-      # Restaure stdout et ferme le descripteur de fichier #6.

echo
echo "== stdout restauré à sa valeur par défaut == "
echo
```

```
ls -al
echo

exit 0
```

Exemple 19.3. Rediriger à la fois `stdin` et `stdout` dans le même script avec *exec*

```
#!/bin/bash
# upperconv.sh
# Convertit un fichier d'entrée spécifié en majuscule.

E_ACCES_FICHIER=70
E_MAUVAIS_ARGS=71

if [ ! -r "$1" ]     # Est-ce que le fichier spécifié est lisible ?
then
  echo "Ne peut pas lire le fichier d'entrée !"
  echo "Usage: $0 fichier-entrée fichier-sortie"
  exit $E_ACCES_FICHIER
fi  #  Sortira avec la même erreur,
    #+ même si le fichier d'entrée ($1) n'est pas spécifié (pourquoi ?).

if [ -z "$2" ]
then
  echo "A besoin d'un fichier de sortie."
  echo "Usage: $0 fichier-entrée fichier-sortie"
  exit $E_MAUVAIS_ARGS
fi


exec 4<&0
exec < $1            # Lira le fichier d'entrée.

exec 7>&1
exec > $2            # Écrira sur le fichier de sortie.
                     #  Assume que le fichier de sortie est modifiable
                     #+ (ajoutez une vérification ?).

# -----------------------------------------------
    cat - | tr a-z A-Z   # Conversion en majuscule.
#   ^^^^^               # Lecture de stdin.
#           ^^^^^^^^^^  # Écriture sur stdout.
# Néanmoins, à la fois stdin et stdout ont été redirigés.
# Notez que le 'cat' peut être omis.
# -----------------------------------------------

exec 1>&7 7>&-       # Restaure stdout.
exec 0<&4 4<&-       # Restaure stdin.

#  Après retour à la normal, la ligne suivante affiche sur stdout comme de
#+ normal.
echo "Le fichier \"$1\" a été enregistré dans \"$2\" après une convertion en majuscule."

exit 0
```

La redirection d'entrée/sortie est un moyen intelligent pour éviter le terrifiant problème des variables inaccessibles à l'intérieur d'un sous-shell.

Exemple 19.4. Éviter un sous-shell

```
#!/bin/bash
# avoid-subshell.sh
# Suggéré par Matthew Walker.

Lignes=0

echo

cat monfichier.txt | while read ligne;
                do {
                  echo $ligne
                  (( Lignes++ )); #  Les valeurs incrémentées de cette variable
                                  #+ sont inaccessibles en dehors de la boucle.
                                  #  Problème de sous-shell.
                }
                done

echo "Nombre de lignes lues = $Lignes"   # 0
                                         # Mauvais !

echo "-------------------------"


exec 3&lt;&gt; monfichier.txt
while read ligne &lt;&3
do {
  echo "$ligne"
  (( Lignes++ ));                 #  Les valeurs incrémentées de cette variable
                                  #+ sont inaccessibles en dehors de la boucle.
                                  #  Pas de sous-shell, pas de problème.
}
done
exec 3&gt;&-

echo "Nombre de lignes lues = $Lignes"     # 8

echo

exit 0

# Les lignes ci-dessous ne sont pas vues du script.

$ cat monfichier.txt

Ligne 1.
Ligne 2.
Ligne 3.
Ligne 4.
Ligne 5.
Ligne 6.
Ligne 7.
Ligne 8.
```

[82] Un *descripteur de fichier* est simplement un numéro que le système d'exploitation affecte à un fichier ouvert pour garder sa trace. Considérez cela comme une version simplifiée d'un pointeur de fichier. C'est analogue à un *handle vers un fichier* en **c**.

[83] Utiliser le *descripteur de fichier 5* pourrait causer des problèmes. Lorsque Bash crée un processus fils, par exemple avec exec, le fils hérite de fd 5 (voir le courrier électronique archivé de Chet Ramey, *SUBJECT: RE: File descriptor 5 is held open*, NdT: Le descripteur de fichier est laissé ouvert). Il est plus raisonnable de laisser ce descripteur tranquille.