Langages de scripts (LS4)

TP 2 – Processus et entrées/sorties

1 Quelques rappels sur les processus et les entrées-sorties

1.1 Processus UNIX

On appelle *processus* un objet dynamique correspondant à l'exécution d'un programme ou d'une commande Unix. Cet objet recense en particulier l'état d'avancement de chaque programme, l'ensemble des données qui lui sont propres, ainsi que d'autres informations sur son contexte d'exécution.

1.1.1 Hiérarchie de processus

Sur les systèmes de type UNIX, tout processus possède les caractéristiques suivantes :
- un nombre entier positif appelé numéro d'identification (PID ou *process identifier*) ;
- l'identifiant du processus qui l'a lancé, ou processus parent, appelé PPID ;
- son propriétaire, en général (mais pas toujours) l'utilisateur qui l'a lancé ;
- éventuellement le terminal dont il dépend (s'il existe) ;
- son répertoire courant, sa priorité de travail, son temps d'exécution, etc.

Chaque nouveau processus est créé à partir d'un processus existant (son *père*), à part le processus `init` de pid 1 dont le processus parent porte par convention le numéro 0 (mais ne correspond pas à un « vrai » processus). Une hiérarchie de processus, dont la racine est `init`, apparaît alors naturellement.

Pour avoir des renseignements sur les processus, on utilise les commandes `ps` et `top`[1]. Par exemple, pour une liste détaillée de tous les processus, on tapera `ps ux`.

Un processus correspondant à un script shell peut connaître son PID grâce à la variable `$`.

1.1.2 Contrôler l'exécution : signaux

Les programmes ne fonctionnent pas toujours comme prévu, il faut pouvoir suspendre ou mettre fin à l'exécution de processus devenus instables ou ne répondant plus. On utilise pour cela un mécanisme de *signaux*, qui permet de façon plus générale d'établir une certaine forme de communication entre processus.

La commande `kill -<sign l> <PID>` permet d'envoyer le `sign l` à un processus dont le PID est connu. Le paramètre `sign l` est soit un nom soit un numéro de signal. Pour connaître la liste des signaux disponibles, on peut utiliser l'option `-l` de `kill`. Le tableau suivant résume les signaux les plus courants, et dans certains cas un raccourci clavier permettant d'envoyer ce signal au processus en cours.

[1] Très lente sur les machines du SCRIPT.

Numéro	Nom	Effet	Raccourci
2	`INT`	Ferme le processus	`Ctrl-C`
9	`KILL`	Tue le processus	
15	`TERM`	Ferme le processus	
17	`STOP`	Suspend le processus	
18	`TSTP`	Suspend le processus	`Ctrl-Z`
19	`CONT`	Relance un processus suspendu	

Les signaux `KILL` et `STOP` ont la particuliarité de ne pas pouvoir être détournés ni ignorés par les processus qui les reçoivent. `TERM` est le signal de terminaison par défaut de `kill`.

Un processus peut modifier sa réaction à un signal grâce à la commande interne `tr p <h ndler> <liste de sign ux>`. Le *handler* est le comportement du processus vis-à-vis du signal. Quand le handler est simple, on pourra se contenter de le mettre directement : `tr p 'echo toto' INT` ; quand il est plus évolué, on passera par une fonction : `tr p m _fonction INT`, cette fonction reçoit un argument qui est le numéro du signal reçu (cf. `kill`).

Fonctions en `b sh` Les fonctions peuvent se déclarer de 2 façons différentes en `b sh` :

```
function nom_fonction {
# les comm ndes
}
```

ou

```
nom_fonction () {
# les comm ndes
}
```

Attention : une fonction doit être déclarée avant d'être appelée[2].

Comme une commande, une fonction est appelée avec des arguments en juxtaposant ses arguments. Elle récupère alors ces derniers grâce aux variables `$1`, `$2`, etc.

1.1.3 Gérer les tâches dans le shell : jobs

Lorsque des processus sont lancés depuis un terminal, `b sh` fournit un ensemble de mécanismes pour gérer leur exécution. On parle de tâches (*jobs*).

Un processus lancé depuis le shell peut se trouver dans trois états distincts :

à l'avant-plan : le processus « occupe » le terminal, et le prompt du shell n'apparaît pas ;

arrêté : le processus a reçu un signal `TSTP` ou `STOP`, son exécution est suspendue ;

à l'arrière-plan : le processus continue à s'exécuter, mais il ne peut pas lire d'entrée sur le terminal.

Une tâche au plus peut s'exécuter à un moment donné en *avant-plan*. Toutes les autres tâches sont soit arrêtées, soit en *arrière-plan*. La commande `jobs` affiche une liste des tâches actuellement en cours d'exécution, ainsi que leur statut. Chaque tâche est associée à un numéro (à ne pas confondre avec le PID !).

[2] On peut contourner cette restriction en utilisant `declare`.

Pour faire passer une tâche arrêtée ou en arrière-plan vers l'avant-plan, on utilise la commande `fg %n` où `n` est le numéro de tâche correspondante (la tâche la plus récente est sélectionnée par défaut).

Pour faire passer une tâche arrêtée vers l'arrière-plan, on utilise la commande `bg` suivie du numéro de processus (la dernière tâche arrêtée est sélectionnée par défaut).

On ne peut pas directement faire passer une tâche de l'avant-plan à l'arrière-plan car le prompt du shell n'est pas actif dans ce cas. Il faut donc auparavant suspendre la tâche à l'avant-plan grâce à la combinaison de touches `Ctrl-Z` ou au signal `STOP`. Il est par contre possible de lancer une commande directement en arrière-plan en la faisant suivre du caractère `&`.

1.2 Redirection d'entrées sorties

Chaque processus peut être amené à manipuler des fichiers au cours de son exécution. A cette fin, il maintient une liste numérotée de 0 à n de *descripteurs de fichiers*, qui recense les fichiers ouverts en lecture ou en écriture par le processus.

Parmi ceux-ci, trois fichiers dits « fichiers standard » occupent une fonction particulière. Ce sont des fichiers seulement au sens logique, c'est à dire qu'ils se comportent comme des fichiers mais peuvent correspondre en réalité à d'autres mécanismes :

- l'entrée standard, qui correspond au descripteur de fichier n°0, où un processus peut conventionnellement recueillir des informations à traiter ;
- la sortie standard, descripteur n°1, où un processus écrit les données qu'il produit ;
- la sortie standard d'erreur, descripteur n°2, où un processus transmet les messages d'erreurs éventuelles rencontrées au cours de son exécution.

Par défaut, ces trois fichiers logiques sont associés au terminal de l'utilisateur (symbolisé par le fichier `/dev/tty`), ce qui permet de lire les données fournies au clavier par l'utilisateur et d'afficher les résultats ou les messages d'erreur sur le terminal.

1.2.1 Redirection de sortie : `>` et `>>`

Le symbole > permet de rediriger la *sortie standard* d'une commande sur un fichier. Cette redirection crée le fichier s'il n'existe pas et l'écrase s'il existe.

Le symbole >> permet également de rediriger la *sortie standard*, mais en cas d'existence du fichier, elle écrit à la suite.

```
comm nde > fichier
comm nde >> fichier
```

1.2.2 Redirection d'entrée : `<`

Les commandes qui attendent un flux d'entrée[3] peuvent le recevoir tapé au fur et à mesure par l'utilisateur (*entrée standard*), ou bien directement d'un fichier grâce au symbole <.

```
comm nde < fichier
```

[3] Attention à ne pas confondre un flux d'entrée et les arguments.

1.2.3 Le canal d'erreur : `2>` et `2>>`

Les messages d'erreur des commandes apparaissent par défaut sur l'écran, tout comme les affichages correspondant aux réponses. Ils arrivent cependant sur un flux différent qui est la *sortie erreur standard*. Le symbole `2>`[4] permet de rediriger cette sortie erreur, avec écrasement du fichier. Le symbole `2>>` fait de même sans écrasement du fichier.

```
comm nde 2> fichier_err
```

1.2.4 Combiner les redirections : `>&`

On peut bien entendu faire plusieurs redirections en même temps :

```
comm nde > fichier 2> fichier_err
comm nde > fichier_out < fichier_in
comm nde < fichier_in > fichier_out
```

etc.

On peut aussi vouloir rediriger un flux sur un autre. Les exemples les plus courants étant de vouloir écrire la sortie standard et la sortie erreur dans le même fichier ou encore de vouloir écrire sur la sortie erreur. Par exemple :

```
comm nde > fichier 2>&1
echo "mess ge d'erreur" >&2
```

1.2.5 Manipuler les descripteurs de fichiers

On peut modifier le flux correspondant à un descripteur avec la commande `exec`.

Si on veut obtenir un descripteur `d` (`d` est un entier) à partir d'un `fichier` donné ou d'un autre descripteur `D`, on fait :

mode d'ouverture	`fichier`	descripteur `D`
lecture	`exec d<fichier`	`exec d<&D`
écriture (avec écrasement)	`exec d>fichier`	`exec d>&D`
écriture (avec ajout en fin)	`exec d>>fichier`	`exec d>>&D`
lecture/écriture	`exec d<>fichier`	`exec d<>&D`
fermeture	`exec d>&-`	`exec d>&-`

Par exemple la commande `exec 2>&1` redirige durablement la sortie erreur standard sur la sortie erreur. Pour récupérer le flux habituel de l'erreur standard, il faut avoir pensé à garder en mémoire un descripteur synonyme de 2 :

```
exec 3>&2  # le descripteur 3 recupere le flux correspond nt   2 en ecriture
exec 2>>errors  # on ouvre le fichier errors en  jout en lui
                #  ttribu nt le descripteur 2

#... code : tous les mess ges d'erreurs sont ecrits d ns le fichier errors

exec 2>&3  # le descripteur 2 recupere le flux correspond nt   3
```

[4] Les symboles précédents >, >>, < sont communs à la plupart des shells, mais il n'en est pas de même pour 2>, on pourra par exemple rencontrer &>.

```
          # c'est- -dire le flux sortie erreur st nd rd
exec 3>&-  # on ferme le descripteur 3

#... code : tous les mess ges d'erreurs sont ecrits sur l  sortie erreur st nd rd
```

De même, la commande `exec 1>fichier` redirige durablement la sortie standard sur un `fichier`. En fait il ouvre le `fichier` en écriture en lui attribuant le descripteur 1.

Dans le tableau ci-dessus, on peut omettre le descripteur `d` s'il s'agit de `0` pour les ouvertures en lecture ou de `1` pour les ouvertures en écriture. Ainsi, `exec >fichier` est équivalent à `exec 1>fichier`.

1.2.6 Les tubes : |

Pour utiliser la sortie d'une commande comme entrée d'une autre commande, sans passer par un fichier régulier intermédiaire, on utlise un *tube*, symbolisé par `|`.

`comm nde1 | comm nde2`

La `comm nde2` reçoit sur son flux d'entrée le flux de sortie de la `comm nde1`.

1.2.7 Le concept de filtre

Les programmes qui traitent des données provenant de l'entrée standard et produisent un résultat sur la sortie standard sont communément appelés « filtres ». Voici quelques exemples typiques de filtres :

- `c t` : recopie sur la sortie ce qu'il reçoit sur l'entrée
- `sort` : trie les données par ordre alphabétique
- `more` : affiche sur la sortie ce qu'il reçoit sur l'entrée, page par page
- `grep` : recherche un motif dans les lignes d'un texte
- `tee` : copie de l'entrée standard sur la sortie standard et sur un fichier

Les filtres sont particulièrement utiles lorsqu'on les combine à l'aide de tubes. Par exemple, pour trier par ordre alphabétique les lignes d'un fichier `liste.txt` puis les afficher page par page, on utilisera :

```
$ c t liste.txt | sort | more
```

2 Exercices

Exercice 1 – *Tuer des processus*

Le but de cet exercice est d'écrire un script de contrôle des processus qui évite d'avoir à manipuler à la main le PID.

1. Écrivez un script `monps.sh` qui affiche un message disant si le nom du processus passé en argument existe. Par exemple, la commande `$ ./monps.sh b sh` doit afficher `b sh est l ncé.`

2. Il est possible que plusieurs processus portant le même nom soient lancés au même moment. Améliorez votre script pour qu'il affiche les PID de tous les processus lancés portant le nom passé en argument.

   Par exemple, si `xclock` a été lancé deux fois avec les PID 24 et 30, la commande `$ ./monps.sh xclock` doit afficher `xclock est l ncé (2 inst nces : 24, 30).`
3. Ajoutez à votre script un paramètre **facultatif** permettant de stopper (`--stop` ou `-s`) ou de tuer (`--kill` ou `-k`) le ou les processus en question, ainsi qu'une option (`--list` ou `-l`) correspondant au comportement par défaut (cf. question précédente). Si aucune option n'est donnée, la commande agira comme si l'option `-l` avait été passée. Le script doit afficher un compte-rendu des processus tués ou arrêtés, ou un message d'erreur si les options ne sont pas correctes.

   Par exemple `$ monps.sh --kill xclock` envoie le signal `KILL` à tous les processus nommés `xclock`.
4. On ne souhaite à présent tuer qu'un seul processus portant un nom donné (par exemple celui qui possède le plus grand PID). Proposez et implémentez une solution.

Exercice 2 – *Tests sur les envois de signaux*

La commande `sleep` suivie d'un entier `n` permet de mettre en sommeil un processus pendant approximativement `n` secondes.

1. Écrire un script qui affiche son PID puis se met en sommeil 100 secondes.
2. À partir d'un autre terminal, envoyez le signal `INT` à votre processus.
3. En sélectionnant les bonnes lignes de `ps` (grâce à un `grep` judicieux), remarquez que votre processus en a engendré un autre. A partir d'un autre terminal, envoyez le signal `INT` à cet autre processus.
4. Relancez votre script et lancez le signal `INT` au groupe de votre processus (voir `m n kill`).
5. Modifiez le comportement de votre script pour qu'il affiche « Je suis mort ! » avant de terminer lorsqu'il reçoit le signal `INT`.

Exercice 3 – *Lecture d'un fichier*

Écrire un script `b sh` qui prend en argument un nom de fichier régulier et qui regarde si le premier "mot" du fichier est en fait un entier. Pour lire dans le fichier, on utilisera une redirection d'entrée.

Exercice 4 – *Un petit démon*

Exercice pouvant compter pour le contrôle continu.

1. a. Dans un script bash, écrivez une *fonction* qui recevra en argument deux entiers représentant respectivement un certain nombre d'heures et un certain nombre de minutes et renverra :
   - `0` si l'heure transmise correspond à l'heure actuelle (commande `d te`),
   - `1` si l'heure transmise se situe plus tard dans la journée,
   - `-1` si l'heure transmise se situe plus tôt dans la journée.

b. Faites en sorte que ce script, une fois lancé avec deux arguments (correspondant à une heure à tester), se mette en sommeil. Il devra “attraper” le signal `INT` et exécuter la fonction précédente avant de se remettre en sommeil.

2. Écrire un autre script qui, une fois lancé, enverra un signal au premier script et affichera alors l'un des messages suivants :
– `top ! c'est fini...` si c'est pile la bonne heure,
– `encore un peu de p tience` s'il reste du temps,
– `vous  uriez pu p rtir depuis longtemps` si l'heure est déjà passée.
Pour celà, vous aurez besoin de mettre en place un moyen de communication entre les deux scripts et donc de modifier le premier (pensez à en faire une sauvegarde avant, surtout si vous voulez le rendre).

Exercice 5 – *Simulation de tube*

Écrire une commande `tube.sh` qui prend en arguments deux commandes `com1` et `com2` et simule `com1 | com2`