



Travaux pratiques informatique
Module Système d'exploitation
Année 2006-2007

TP N°2 : Scripts avancés & Processus & Signaux

Mathieu.Barcikowski@univ-savoie.fr

Pierre.Hyvernats@univ-savoie.fr

Cyril.Vachet@etu.univ-savoie.fr

1. Présentation

Ce TP a pour objectif de vous faire appréhender les structures de script avancés, certaines notions sur les processus ainsi que des signaux en Shell. Ce travail donnera lieu à un compte-rendu et à des réalisations qui seront notés.

2. Instructions de Rendu

(1) Pour le compte-rendu, vous devrez mettre les objectifs, les problématiques du TP et comment vous y avez répondu. Pour certains problèmes, il sera par ailleurs nécessaire de mettre en avant les concepts du cours que vous aurez appliqué. La clarté et la propreté du compte rendu est un élément de base à la notation.

(2) Pour les résultats, le TP qui vous ai proposé est un sujet relativement ouvert. Votre objectif premier est d'appréhender l'ensemble des questions tout en vous concentrant sur les points essentiels : pour chaque question, il ne vous est pas forcément demandé de réaliser « le programme parfait » mais plutôt de montrer à votre correcteur que vous avez acquis certains savoirs essentiels. De ce fait, la bonne stratégie est de réaliser des embryons de solutions fonctionnels puis de les améliorer par la suite en dehors des horaires du TP.

3. Scripts Shell Avancés (1h30)

Dans cette partie, on vous demande de « surcharger » une commande du générique du shell. Il s'agit de la commande « rm » qui permet de supprimer des fichiers. Cette commande gère déjà une multitude d'options mais n'intègre pas de fonctionnalités de « corbeille » : lorsqu'un fichier est supprimé, ce dernier l'est pour toujours; il n'est pas possible de le restaurer. Ainsi, nous allons réaliser la commande « superrm » qui aura des options similaires à la fonction « rm » mais qui sauvegardera, avant suppression, une copie du fichier dans le répertoire « rm_trash ».

Attention : Les commandes « rm » et « superrm » sont dangereuses pour votre environnement. Veillez à travailler sur des fichiers ainsi que 1 ou des répertoires de test !

Options possibles de la commande « superrm »:

Option	Signification	Provenance
-f ou --force	Ne pas interroger l'utilisateur.	/bin/rm
-i ou --interactive	Interroger l'utilisateur avant de supprimer le fichier	/bin/rm
-r, -R ou --recursive	Supprimer de manière récursive les répertoire / fichier.	/bin/rm
-v ou --verbose	Afficher les noms de fichier avant de les supprimer.	/bin/rm
--help	Affiche une page d'aide	/bin/rm
--version	Affiche la version de la commande	/bin/rm
-e ou --empty	Vide le contenu de la corbeille de sauvegarde.	rm_trash
-l ou --list	Liste le contenu de la corbeille de sauvegarde.	rm_trash
-s ou --restore	Récupère un fichier dans la corbeille (// à son nom) et le copie à l'endroit où est tapé la commande super_rm.	rm_trash
-d ou --directory	Supprime le dossier	/bin/rm

L'analyse de l'algorithme du « superrm » est très simple, ce dernier se décompose en deux sous étapes :

- **Traitement des options** : consiste à marquer l'ensemble des options qui ont été sélectionnées.
- **Exécution de la commande en fonction des options sélectionnées** : fait appel aux commandes shell déjà existantes pour obtenir un résultat conforme aux options sélectionnées par l'utilisateur.

Conseils :

- Un choix judicieux est de déclarer des variables locales afin de gérer l'état des options sélectionnées puis de réutiliser ces options (test) lors de l'exécution effective de la commande..
- L'utilisation de la commande **getopts** permet de faciliter la gestion des options (voir aussi \$OPTIND, shift etc.).
- L'utilisation de la structure « case » facilite la gestion d'option multiples.
- Faites attentions aux options incompatibles entre elles !!

4. Les processus (2h30)

On vous demande d'écrire le programme C suivant puis le compiler à l'aide de gcc :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (void)
{
    pid_t pid ;
    if ((pid = fork()) < 0)
    {
        fprintf (stderr, "échec du fork()\n");
    }
    if (pid != 0)
    {
        /* processus père */
        fprintf (stdout, "père : mon PID est %u.\n", getpid()) ;
        sleep(100) ;
        fprintf (stdout, "père : je me termine...\n");
        exit (0) ;
    }
    else
    {
        /* processus fils */
        fprintf (stdout, "fils : mon PID est %u.\n", getpid());
        sleep(1) ;
        fprintf (stdout, "fils : je me termine...\n");
        exit(0) ;
    }
    return(0);
}
```

Question :

- A l'aide des commandes spécifiques (ps, pstree) et le résultat de l'exécution de ce programme, essayez d'expliquer ce qu'il se passe ? Quels sont les mécanismes du système pour gérer ce type de problème ?
- Rajoutez le code suivant au niveau du processus fils. Que se passe t'il ?

```
pid2 = fork();
if (pid2 != 0)
{
    /* processus fils */
    fprintf (stdout, "fils : mon PID est %u.\n", getpid());
    sleep(1) ;
    fprintf (stdout, "fils : je me termine...\n");
    exit (0) ;
}
else {
    fprintf (stdout, "fils du fils : mon PID est %u.\n", getpid());
    sleep(50) ;
    fprintf (stdout, "fils du fils : je me termine...\n");
    exit (0) ;
}
```

Question (ouverte) :

- Par expérimentation successive et par recherche documentaire, débattre de manière plus générale et en quelques lignes des mécanismes du `fork()` et généraliser sur son utilisation.
- Débattre de manière plus générale et en quelques lignes de la différence entre un processus lourd (programme, `fork`) et un processus léger (thread `posix`). Quels sont leurs avantages et inconvénients respectifs ? Quels sont les problèmes de sécurité liés à l'utilisation d'un `fork` ?

5. Introduction aux mécanismes des signaux

Nous souhaitons appréhender les mécanismes des signaux sous Linux. Pour cela, nous allons utiliser un ensemble de commandes shell spécifiques. Testez le script suivant :

```
trap "echo Le script s'est terminé" EXIT
trap "echo Vous avez appuyé sur Ctrl-C" SIGINT
trap "echo Vous avez fait: kill $$" SIGTERM
trap "echo J'ai été stoppé et je continue" SIGCONT
trap "echo J'ai reçu SIGUSR1 ou SIGUSR2" SIGUSR1 SIGUSR2
echo "Processus $$: J'attends un signal..."
#
# Compte à rebours
#
i=100
while [ $i -gt 0 ]; do
echo -n "$i "
sleep 1
let $[ i -= 1 ]
done
echo "0"
```

On vous demande de réaliser en parallèle sur un autre terminal les commandes suivantes :

- `Ctrl-z` puis `fg`,
- `Ctrl-c`,
- `kill -9 numpid`,
- `kill -s SIGUSR1 numpid`,
- `kill -s SIG(n'importe quel signal) numpid`.

Question :

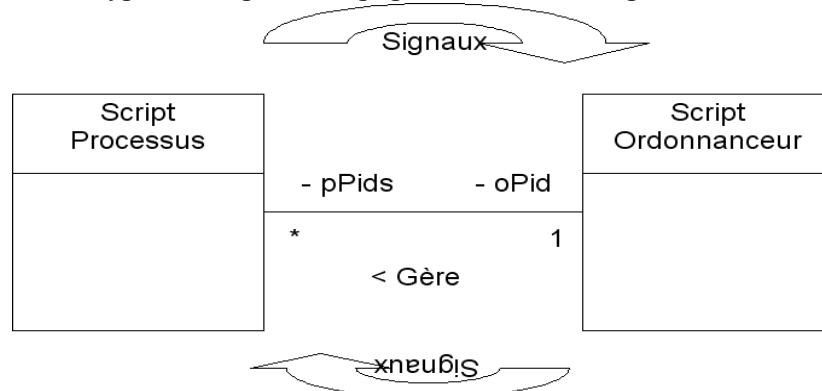
- Déterminez ce que fait ce script.
- A quoi sert le `trap` ?
- Quels sont les fonctionnalités relatives à la commande `kill` ?

6. Ordonnement simple : simulation à l'aide du Shell

Dans cette question, on vous demande de réaliser un ordonnanceur à l'aide du système des signaux sous Linux. Pour réaliser cet ordonnanceur, nous utiliserons les stratégies d'ordonnement abordées lors du TD1.

Le principe de l'ordonneur peut être schématisé de la manière suivante :

Nous avons deux type de script, le script processus et le script ordonnanceur, qui s'envoie



mutuellement des signaux (produisent des interruptions) dans le but de se synchroniser. Ces scripts peuvent être décrits de la manière suivante :

- **Script Processus** : Il s'agit d'un script générique représentant un processus. Il a un numéro « pid » généré par le système et il connaît le numéro « pid » de l'ordonneur. Il a par ailleurs et à des propriétés judicieusement choisies pour faciliter son ordonnancement ainsi qu'un corps : un ensemble de commande qu'il doit exécuter. Dans un but simplificateur, le corps du processus sera chargé d'afficher la valeur d'un compte à rebours affichant la quantité de quantum restant à exécuter.
- **Script Ordonneur** : Il s'agit du script chargé de l'ordonnement des différents processus dont il connaît le pid. L'ordonnement est réalisé selon une certaine stratégie.

Scénario possible :

1a. Lancement manuelle : chaque instance de script est lancé à la main. L'utilisateur est chargé de donner les options et faire l'échange des différents pids à l'aide de saisies clavier.

1b. Lancement semi-automatique : l'ordonneur lance un ensemble déterminé de processus. Les différents pid sont échangés dans le script.

2. Ouverture des terminaux correspondant à chacun des scripts qui ont été lancés.

3. Visualisation de l'ordonnement dans chacun des terminaux.

Question :

- On vous demande d'implémenter cet ordonnanceur en utilisant comme stratégie le tourniquet sans préemption avec un quantum de temps égal à n.

- Définir (sans coder) les grandes lignes de votre approche pour prendre en compte les autres stratégies d'ordonnancement.

Conseil : n'oubliez pas de mettre en évidence vos choix tout en les argumentant dans votre compte rendu.

Question (ouverte) :

- En quelques lignes, mettez en avant les limites de cette simulation par rapport à ce qu'il se passe dans la réalité (manques, simplifications,...).

Question (de cours) :

- Débattez en quelques lignes des avantages mais aussi des inconvénients des systèmes d'ordonnancement évolué par rapport au traitement par lot.