Systèmes d'exploitation

TP 2 - Processus

2023-03-06

1 Contrôle des processus

Cet exercice est librement inspiré du cours Job control fourni par la Software Carpentry Foundation et disponible sous licence CC-BY 4.0.

Nous allons voir comment contrôler les programmes *une fois qu'ils sont en cours d'exécution*. Lorsque nous parlons de contrôler des programmes, ce que nous voulons vraiment dire est contrôler les *processus*. Comme nous l'avons vu dans le cours, un processus est un programme qui est en mémoire et qui s'exécute. Certains des processus de votre ordinateur sont les vôtres : ils exécutent des programmes que vous avez explicitement demandés, comme votre navigateur Web. De nombreux autres appartiennent au système d'exploitation qui gère votre ordinateur ou, si vous êtes sur une machine partagée, à d'autres utilisateurs.

1.1 La commande `ps`

Vous pouvez utiliser la commande `ps` pour lister les processus, tout comme vous utilisez `ls` pour lister les fichiers et répertoires.

```
$ ps
```

Combien de processus actifs voyez-vous ?

Au moment où vous avez exécuté la commande `ps`, vous aviez au moins deux processus actifs, votre interpréteur de commandes (`bash`) et la commande (`ps`) que vous aviez invoquée dans celui-ci.

Maintenant, affichons davantage d'informations avec la commande suivante :

```
$ ps -f
```

L'option `-f` signifie "liste complète" (*"full-format listing"*).

Mais que nous dit-on ici ?

- Chaque processus a un **identifiant unique (PID)**. Rappelez-vous, c'est une propriété du processus, pas du programme que ce processus exécute : si vous exécutez trois instances de votre navigateur en même temps, chacune aura chacune aura son propre identifiant de processus.
- La colonne **PPID**, indique l'ID du parent de chaque processus. Chaque processus d'un ordinateur est engendré par un autre processus, qui est son parent (à l'exception, bien entendu, du processus d'amorçage qui s'exécute automatiquement au démarrage de l'ordinateur).
- La colonne **UID** montre le nom de l'utilisateur qui exécute les processus.
- La colonne **STIME**, indique quand le processus a commencé à s'exécuter.
- La colonne **TIME**, vous montre combien de temps le processus a utilisé.
- La colonne **CMD** montre quel programme le processus est en train d'exécuter.
- La colonne **TTY**, indique l'ID du terminal dans lequel le processus est exécuté.
- La colonne **C** est une indication du pourcentage d'utilisation du processeur.

Différentes versions de `ps`

Votre version de `ps` peut peut afficher plus ou moins de colonnes, ou les afficher dans un ordre différent, mais les mêmes informations sont généralement disponibles partout et les en-têtes de colonnes sont généralement cohérents.

Comportement de la commande `ps`

La commande `ps` possède un grand nombre d'options qui contrôlent son comportement et, de plus, les jeux d'options et le comportement par défaut varient selon les plateformes.

Une invocation simple de `ps` ne vous montre que des informations de base sur vos processus *actifs*.

Comment lister tous les processus du système avec `ps` ? Comment lister tous les processus d'un utilisateur donné ?

Vous pouvez afficher les processus sous forme d'arborescence grâce à l'option `--forest` :

```
$ ps -e --forest
```

1.2 Arrêter, mettre en pause, reprendre et mettre en arrière-plan des processus

L'interpréteur de commandes fournit plusieurs commandes permettant d'arrêter, de mettre en pause et de reprendre les processus.

Pour les voir en action, commençons par créer un script `calculer.sh` pour simuler un long calcul et afficher une barre de progression. Dans votre éditeur de texte Nano, entrez les commandes suivantes (ne vous souciez pas trop du contenu du fichier) :

```
#!/bin/bash

count=0
total=1000
pstr="[=========================================================================]"

while [ $count -lt $total ]; do
  sleep 0.5 # this is work
  count=$(( $count + 1 ))
  pd=$(( $count * 73 / $total ))
  printf "%3d.%1d%% %.${pd}s\n" \
    $(( $count * 100 / $total )) $(( ($count * 1000 / $total) % 10 )) $pstr
done
```

Rendez le script exécutable.

```
$ chmod +x calculer.sh
```

Lançons maintenant notre programme `calculer.sh`.

```
$ ./calculer.sh
```

Après quelques minutes, nous réalisons que cela va prendre un certain temps pour se terminer. Stoppons ("tuons") le processus en, en tapant `CTRL` + `C`. Cela arrête immédiatement le processus en cours d'exécution.

Exécutons à nouveau la même commande, avec une esperluette `&` à la fin de la ligne pour indiquer au shell que le programme doit s'exécuter en arrière-plan (mode *background*).

```
$ ./calculer.sh &
```

Lorsque nous faisons cela, le shell lance le programme comme précédemment. Au lieu de au lieu de laisser notre clavier à attaché à l'entrée du programme, l'interpréteur de commandes s'y accroche. Cela signifie que le shell peut nous donner une nouvelle invite de commande, et commencer à exécuter d'autres commandes, immédiatement.

Redirection de la sortie et de l'erreur standards

Par défaut, l'exécution d'une commande en arrière-plan, avec `&`, ne détache que `stdin` (l'entrée standard), cependant `stdout`(la sortie standard) et `stderr` (l'erreur standard) demeurent attachés au shell parent.
Si vous ne voulez pas voir `stdout` et/ou `stderr`, vous pouvez les rediriger vers un fichier ou vers `/dev/null` (périphérique null) :

```
# redirige stdout et stderr vers log.txt
./calculer.sh &>log.txt &

# redirige stdout vers /dev/null
./calculer.sh 1>/dev/null &

# redirige stdout vers output.log, stderr vers error.log
./calculer.sh 1>output.log 2>error.log &
```

Maintenant, lançons la commande `jobs`, qui nous indique les processus qui s'exécutent actuellement en arrière-plan :

```
$ jobs
```

Puisque nous sommes sur le point d'aller chercher un café, nous pourrions tout aussi bien utiliser la commande d'avant-plan, `fg`, pour mettre notre travail d'arrière-plan au au premier plan :

```
$ fg
```

Quand `./calculer.sh` se termine, l'interpréteur de commandes nous donne un nouveau prompt comme comme d'habitude.

Sélectionner le processus à mettre au premier plan

Si nous avions plusieurs travaux en cours d'exécution en arrière-plan, nous pourrions contrôler lequel mettre au premier plan en utilisant `fg %1`, `fg %2`, et ainsi de suite. Les ID ne sont *pas* les ID des processus. Ce sont plutôt les identifiants des tâches affichés par la commande `jobs`.

L'interpréteur de commandes nous donne un outil supplémentaire pour contrôler les travaux déjà en cours d'exécution au premier plan. La combinaison de touches `CTRL` + `Z` mettra le processus en pause et rendra la main à l'interpréteur de commandes. Nous pourrons alors utiliser `fg` pour le reprendre en avant-plan, ou `bg` pour le reprendre en arrière-plan.

Par exemple, lançons à nouveau `./calculer.sh`, puis tapons `CTRL` + `Z`. Le shell nous indique immédiatement que notre programme a été arrêté, et nous donne son numéro de job :

```
[1]  + 12668 suspended  ./calculer.sh
```

Si nous tapons `bg %1`, l'interpréteur de commandes relance le processus en cours d'exécution, mais en arrière-plan. Nous pouvons vérifier qu'il est bien en cours d'exécution en utilisant `jobs`.

Nous pouvons ensuite l'arrêter ("le tuer") pendant qu'il est encore en arrière-plan en utilisant `kill` et le numéro du job. Cela a le même effet que de le faire passer au premier plan, puis de taper `CTRL-C` :

```
$ kill %1
```

Signaux

La commande `kill` est utilisée pour envoyer des signaux à un processus, notamment des directives de pause, d'arrêt et d'autres directives utiles.

Par commodité, dans la plupart des shells Unix, certaines combinaisons de touches sont configurées pour délivrer un signal spécifique au processus en cours d'exécution. Par exemple, `CTRL + C` envoie un signal `SIGINT` (interruption) au processus (ce qui met normalement fin à l'exécution) et `CTRL + Z` envoie un signal `SIGTSTP` (stop), interrompant ainsi le processus en cours d'exécution.

2 Créer des processus avec l'API Unix

Cet exercice provient de l'ouvrage ***Operating Systems: Three Easy Pieces*** *de Remzi H. Arpaci-Dusseau et Andrea C. Arpaci-Dusseau.*

Programmer en C

Si vous n'êtes pas familier avec la compilation de programme en C, consultez ce tutoriel (en anglais). Il contient des conseils utiles pour la programmation dans l'environnement C.

Documentation des appels systèmes

Vous pouvez accéder à la documentation des appels systèmes Unix grâce à la commande `man`. Par exemple, la documentation de `fork()` est accessible avec la commande `man 2 fork`.

Vous pouvez également retrouver les *man pages* en ligne sur man7.org.

Exemples

Pour des exemples d'utilisation de `fork()`, `exec()` ou `wait()`, référez vous au chapitre Process API.

Cet exercice va vous permettre de vous familiariser avec les API de gestion des processus d'Unix.

1. Écrivez un programme qui appelle `fork()`. Avant d'appeler `fork()`, demandez au processus principal d'accéder à une variable (par exemple, x) et de lui donner une valeur (par exemple, 100). Quelle est la valeur de la variable dans le processus enfant ? Qu'arrive-t-il à la variable lorsque l'enfant et le parent changent la valeur de x ?
2. Écrivez un programme qui ouvre un fichier (avec l'appel système `open()`) et qui appelle ensuite `fork()` pour créer un nouveau processus. L'enfant et le parent peuvent-ils tous deux accéder au descripteur de fichier renvoyé par `open()` ? Que se passe-t-il lorsqu'ils écrivent dans le fichier simultanément, c'est-à-dire en même temps ?
3. Écrivez un autre programme en utilisant `fork()`. Le processus enfant doit imprimer "hello" ; le processus parent doit imprimer "goodbye". Vous devez essayer de faire en sorte que le

processus enfant imprime toujours en premier ; pouvez-vous le faire sans appeler `wait()` dans le processus parent ?

4. Écrivez un programme qui appelle `fork()` et qui appelle ensuite une forme de `exec()` pour exécuter le programme `/bin/ls`. Voyez si vous pouvez essayer toutes les variantes de exec(), y compris (sous Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()` et `execvpe()`. Pourquoi pensez-vous qu'il existe autant de variantes du même appel de base ?

5. Maintenant, écrivez un programme qui utilise `wait()` pour attendre que le processus enfant se termine dans le parent. Que retourne `wait()` ? Que se passe-t-il si vous utilisez `wait()` dans le processus enfant ?

6. Écrivez une légère modification du programme précédent, en utilisant cette fois-ci `waitpid()` au lieu de `wait()`. Quand `waitpid()` serait-il utile ?

7. Écrivez un programme qui crée un processus enfant, et ensuite dans l'enfant ferme la sortie standard (`STDOUT_FILENO`). Que se passe-t-il si l'enfant appelle `printf()` pour imprimer une sortie après avoir fermé le descripteur ?

8. Écrivez un programme qui crée deux enfants et connecte la sortie standard de l'un à l'entrée standard de l'autre, en utilisant l'appel système `pipe()`. Vous aurez également besoin de l'appel système `dup2()`.

À retenir

- Les informations sur les processus peuvent être inspectées avec `ps`.
- Chaque processus possède un identifiant unique nommé *process ID* (PID).
- `&` permet d'exécuter une commande en arrière-plan.
- `jobs` affiche tous les processus qui sont actuellement suspendus ou en cours d'exécution en arrière-plan.
- Les processus mis en pause ou en arrière-plan peuvent être ramenés au premier plan avec `fg`.
- `CTRL` + `C` met fin à l'exécution du processus.
- `CTRL` + `Z` met le processus en pause.
- Le contrôle du processus est disponible sous la forme de signaux, qui peuvent causer la pause, la reprise ou même la fin d'un processus.
- L'appel système `fork()` est utilisé dans les systèmes Unix pour créer un nouveau processus. Le créateur est appelé le parent ; le processus nouvellement créé est appelé l'enfant. appelé l'enfant. Le processus enfant est une copie presque identique du processus parent.
- L'appel système `wait()` permet à un parent d'attendre que son enfant termine son exécution.
- La famille d'appels système `exec()` permet à un enfant de s'affranchir de sa similitude avec son parent et d'exécuter et d'exécuter un programme entièrement nouveau.