TP de Systèmes d'exploitation
François Cayre / francois.cayre@grenoble-inp.fr

# Démons et signaux

Notions abordées : création des processus, gestion des signaux.

Fonctions utilisées : `fork(2)`, `setsid(2)`, `system(3)`, `getpid(2)`, `atexit(3)`, `exit(3)`, `signal(2)`, `kill(1,2)`, `pause(2)`.

## 1 Services d'un système d'exploitation

Un système d'exploitation est là pour rendre des services à l'utilisateur. Ces services incluent, par exemple, la gestion d'une file d'impression (serveur d'impression), la distribution de pages *web* (serveur `http`), ou encore la mise à jour régulière et automatique des logiciels installés.

Un système d'exploitation évolue dans différents modes (*runlevels*) : par exemple, en mode maintenance on va vouloir désactiver les services inutiles pour procéder à une mise à jour importante... Pour ensuite revenir en mode normal (en production). C'est `systemd(1)` qui gère les services sous Linux, à travers le processus de PID 1, `init(1)`, qui est l'aïeul de tous les autres processus. Lorsqu'un processus père meurt avant son/ses processus fils, c'est `init(1)` qui en devient automatiquement le nouveau père. Ainsi, `init(1)` peut facilement demander l'arrêt de ses fils en leur envoyant le signal `SIGTERM`. Les services sont généralement rattachés à `init(1)`.

Ces services :

1. ne peuvent exister en double dans le système ;
2. ne doivent répondre que lorsqu'il y en a besoin (notion d'événement) ;
3. doivent être pilotés par le système d'exploitation à travers un ensemble uniformisé de commandes (`start`, `stop`, `restart`, `status`, *etc.*) ;
4. tiennent à jour un fichier de journal (fichier dit de *log*), retraçant les événements de la vie du service (souvent dans `/var/log`) ;
5. sont adaptés aux besoins de l'utilisateur à travers un fichier de configuration (souvent dans `/etc`).

*Corollaire* : La plupart des serveurs (`http`, `ftp`, `ssh`, impression, etc.) sont implantés sous forme de service. Par exemple, une fois le fichier `/etc/apache24/httpd.conf` débogué en ayant trouvé le problème dans `/var/log/apache24/error.log`, un administrateur pourra redémarrer manuellement un serveur *web* Apache 2.4 sous un Linux avec :

```
# systemctl restart apache24
```

Dans un système d'exploitation, un service est généralement implanté sous la forme d'un *démon*.

# 2 Démons

Les processus dont vous avez l'habitude sont pilotés par le clavier (*e.g.*, en appuyant sur `Ctrl-c`, vous pouvez envoyer le signal `SIGINT` à un programme dans une boucle infinie pour demander l'action par défaut associée à ce signal : l'arrêt du processus.)

Un démon est piloté uniquement par des signaux (il n'est pas relié au clavier). Lorsqu'il n'a pas de signal à traiter, il est en sommeil et attend un signal. Son père est `init(1)`.

On dit qu'un démon tourne *en tâche de fond*. Exactement comme lorsque vous rajoutez un `&` après une commande dans le *shell* (différence : votre processus n'est alors pas rattaché à `init(1)`).

La séquence canonique pour lancer un démon en tâche de fond est la suivante :

1. Premier `fork(2)`
2. Dans le fils : déconnexion du clavier par `setsid(2)`
3. Toujours dans le fils : second `fork(2)`
4. Dans le (petit-)fils : lancement de la boucle principale du code du démon.

*Explication* : Les processus vivent en groupes, appelés *sessions* de processus. Une *session* est associée à un terminal de contrôle (quel terminal peut envoyer des caractères sur l'entrée standard des processus de la session ?) Tous les processus descendants d'un même père sont dans la même session, et la fonction `setsid(2)` permet de créer une nouvelle session à partir du processus qui l'appelle. Cette nouvelle session ne dispose d'aucun terminal de contrôle. Remarquez que les pères n'attendent pas la terminaison des fils pour mourir. Ainsi, les fils sont automatiquement rattachés à `init(1)`, qui peut les piloter. *In fine*, le petit-fils sera seul dans sa session, sans terminal de contrôle, et rattaché à `init(1)`. Ce qui est exactement ce que nous voulions.

Un œil aguerri aura bien vite remarqué que le second `fork(2)` semble superflu. En réalité, l'étape 2 inclut souvent d'autres tâches qui le rendent nécessaire, sur lesquelles nous ne nous étendons pas, et nous suivrons donc la manière standard de créer un démon en tâche de fond.

# 3 Ce petit `cron` est un vrai démon

Il existe quantité de tâches à réaliser automatiquement sur un système, souvent à intervales réguliers (lancer les sauvegardes le vendredi soir, installer les mises à jour tous les soirs, *etc.*) Un administrateur a donc besoin d'un service permettant d'exécuter une commande au moment voulu. Sur un système Unix, c'est le démon `cron(8)` qui est chargé de cette tâche, et il est piloté par le fichier de configuration `/etc/crontab`.

Nous allons reprogrammer une version très simplifiée de `cron(8)`, qui lancera une commande à une heure donnée. Comme nous ne sommes pas administrateur des machines de TP, nous n'allons pas pouvoir utiliser les répertoires `/var/log` et `/etc`. Nous appellerons notre démon `crond`.

### 3.1 Fichier de verrou de `crond` : `cron.lock`

Afin de garantir qu'un seul `crond` tourne dans la machine, nous le programmerons pour qu'il se termine par un échec s'il détecte la présence d'un fichier de verrou (*lock file*), sinon il crée ce fichier de verrou (`cron.lock`), il y écrit son PID à l'intérieur, et termine de se lancer. De cette manière, un processus quelconque sait à quel PID envoyer un signal pour piloter `crond`.

### 3.2 Fichier de configuration de `crond` : `cron.conf`

Les heures et commandes à exécuter à ces heures sont stockées dans le fichier cron.conf. Nous vous fournissons de quoi récupérer et gérer une liste chaînée d'événements (heure et commande) à traiter dans le futur à partir de `cron.conf`.

### 3.3 Fichier de journal de `crond` : `cron.log`

Notre démon tient à jour le fichier cron.log, en y ajoutant des lignes de la forme :

```
==PID== {toto.c:42} [HEURE]  Il s'est passé tel truc !
```

Nous vous fournissons de quoi tenir un tel fichier de journal. En particulier, la macro `LOG(message)` s'utilise comme `printf(3)` pour afficher le `message` à la fin de `cron.log`.

### 3.4 Interface de commande, signaux

Notre `crond` reconnaîtra les commandes suivantes, éventuellement associée à un signal dédié à envoyer au démon dont le PID est stocké dans `cron.lock` :

| Commande | Signal associé | Spécification |
|---|---|---|
| `$ ./crond help` | Néant | Affiche l'aide. |
| `$ ./crond start` | Néant | Démarre le démon `crond` si `cron.lock` n'existe pas. |
| `$ ./crond stop` | `SIGTERM` | Termine le démon `crond` proprement. |
| `$ ./crond restart` | Néant | Termine le démon `crond` proprement s'il existe, puis crée un nouveau démon `crond`. |
| `$ ./crond status` | `SIGUSR1` | Affiche la liste des commandes restant à exécuter. |
| `$ ./crond reload` | `SIGINT` | Vide la liste des commandes restant à exécuter, recharge le fichier `cron.conf` et positionne la première alarme. |

Par *terminer proprement*, il faut comprendre : vider la liste des commandes restant à exécuter, effacer le fichier de verrou, et fermer le fichier de journal. Toutes ces actions sont à exécuter au moment d'appeler `exit(3)`, enregistrez-les (dans le bon ordre !) avec `atexit(3)`.

Le démon gère pour lui-même le signal `SIGALRM` qui le prévient qu'il est l'heure d'exécuter la prochaine commande à l'aide de la fonction `system(3)`.

## *3.5 Déboguer depuis le terminal*

Puisque notre démon ne peut plus être piloté par aucun clavier, il faut donc pouvoir lui envoyer des signaux depuis le terminal. Pour lui envoyer le signal `SIGTERM`, on utilisera `kill(1)` et le fichier de verrou :

```
$ kill -SIGTERM `cat cron.lock`
```

Bon travail !