Programmation d'un démon Unix

Lionel Tricon - lionel.tricon@free.fr

Transformer un processus ordinaire en démon est un des fondamentaux incontournables de toute programmation Unix. Cet article se propose de balayer toutes les étapes nécessaires pour une telle transformation tout en tenant compte des impératifs de sécurité inhérents à la constitution d'un tel programme par principe plus vulnérable aux attaques externes (un démon fait généralement office de serveur).

Article publié dans le numéro 72 de Linux Magazine France (mai 2005).

Du point de vue de l'utilisateur, un système d'exploitation semble uniquement peuplé d'applications graphiques de type navigateur, lecteur de mail ou encore traitement de texte. Physiquement, le dialogue avec la machine s'établit par l'intermédiaire du clavier, de la souris et de l'écran ; ces programmes sont dit interactifs car ils interagissent justement avec des entités pensantes (les utilisateurs).

Ce type de programmes n'existe que le temps d'une session : ils sont exécutés puis arrêtés par la volonté de l'utilisateur et ne survivent pas à la déconnexion de la session ouverte sur la machine. On voit que cela ne convient visiblement pas si l'on souhaite laisser un processus serveur (ie. qui rend un service à d'autres programmes) tourner sans discontinuer sur une machine.

Il existe donc une famille de processus, appelés « **démons** », dont le rôle est justement de tourner silencieusement sur le système, en tâche de fond, sans interface avec le moindre utilisateur ou la moindre console.

D'un point de vue système, les programmes sont de simples processus qui possèdent des droits, des flux d'entrées-sorties, des descripteurs de fichiers et différents mécanismes d'héritage des propriétés entre les processus pères et les processus fils (signaux, session, ...). La notion d'interface graphique n'apparaît pas à ce stade là car elle est déportée sur un média extérieur comme par exemple un serveur X : Un démon est donc un simple processus ordinaire que l'on va isoler judicieusement du système pour le rendre autonome.

En mode commande sous Unix, lorsque l'on lance un programme dans une console (un émulateur de terminal), celui-ci reste « accroché » car il va s'en servir comme terminal de contrôle (la console devient le père du programme) : cela peut être très pratique dans la phase de débogage mais pose problème lorsque l'on souhaite le détacher du terminal.

Pour transformer un processus ordinaire en démon, il va falloir justement couper tout lien avec le processus appelant (un terminal ou un autre processus) car lorsque l'on crée un nouveau processus, on hérite logiquement de tous les attributs du père auquel on reste lié (flux stdout, stdin et stderr inclus) : En particulier, si le père meurt, le fils meurt (un programme envoie un signal SIGHUP à tous ses processus fils lorsqu'il reçoit lui aussi ce signal). Pas très pratique lorsque le but est justement de rendre notre programme autonome.

Généralement, ce type de processus est lancé dans la phase d'initialisation du système et demeure résident jusqu'à l'arrêt du système ou sur action opérateur (d'où son surnom de « processus résident »).

///Début encadré///

Il existe une autre façon de rendre l'exécution d'un script ou d'un exécutable insensible à toute déconnexion de la session courante par l'intermédiaire du programme « **nohup** » (contraction de « no hang up »). Ce utilitaire exécute la commande passée en argument en la rendant insensible aux signaux SIGHUP (ceux traitant de la déconnexion) et en changeant son niveau de priorité à 5 ; la redirection de la sortie standard et de la sortie erreur se fait soit dans un fichier explicitement spécifié par l'utilisateur, soit dans le fichier nohup.out si la sortie standard est un terminal (tty). On peut forcer l'exécution à se faire en arrière-plan en ajoutant le caractère « & » à la fin de la ligne de commande.

///Fin encadré///

Du fait qu'elle offre des services (le plus souvent réseau), cette famille de processus est généralement plus vulnérable aux attaques externes. Il faut donc apporter un intérêt tout particulier à la conception du service et à la façon dont nous allons le faire passer en tâche de fond : les droits de création des fichiers, les répertoires accessibles sur la machine, l'utilisateur sous lequel tourne le démon sont autant de failles potentielles qu'il va falloir anticiper.

Bien qu'un processus résident ne possède pas d'interface à proprement dite avec les utilisateurs, en dehors du service rendu, il est plus que conseillé de tracer dans un fichier certains évènements utiles à l'exploitation (messages d'erreur, de debug ou simples informations pratiques). Il doit aussi pouvoir réagir à l'envoi de signaux, par exemple pour le forcer à mourir proprement ou à relire un fichier de configuration.

Un serveur de mail est un exemple concret de processus résident qui offre un service réseau transparent aux utilisateurs internes ou externes du système qui ignorent parfois même jusqu'à son existence.

Pour résumer, les démons sont des processus ordinaires qui possèdent les particularités suivantes :

- Ils ne sont pas rattachés à un terminal particulier (ils tournent en tâche de fond),
- Ils sont indépendants et n'héritent d'aucun attribut du père,
- Ils sont le plus souvent lancés au démarrage du système,
- Ils ne permettent généralement qu'une seule instance d'exécution,
- Ils s'exécutent généralement avec des droits restreints,
- Ils ne s'arrêtent en principe jamais (sauf arrêt de la machine).

Mise en pratique

Les sections suivantes présentes les différentes étapes nécessaires pour transformer un processus ordinaire écrit en langage C/C++ en démon. Certaines sont obligatoires, d'autres optionnelles ; à vous de faire votre choix.

Création d'un processus fils

Nous allons dupliquer le processus appelant puis libérer ce dernier grâce à l'appel-système **`fork()`**. Cette primitive autorise la création d'un processus fils qui ne diffère de son père que par les points suivants :

* Nouveau numéro de processus pour le fils (PID),

* On associe le processus appelant comme processus père (PPID),

* Les statistiques d'utilisation des ressources sont remisent à zéro (utime, stime, ...),

* Les verrouillages des fichiers ne sont pas transmis,

* Les signaux en attente ne sont pas hérités.

```
int pid = fork();
if (pid < 0) { perror("echec fork"); exit(-1); }
if (pid > 0) { exit(0); /* nous sommes dans le père */ }
/* nous sommes dans le fils */
```

Création d'une nouvelle session

Il nous faut maintenant créer une nouvelle session pour le processus grâce à la primitive **`setsid()`** afin de le détacher du père et de son éventuel terminal de contrôle. On peut valider le fait qu'un programme possède un terminal de contrôle en consultant le champ TTY de la commande « **`ps aux`** ».

Sous Unix, tous les processus appartiennent à un groupe : les groupes de processus sont utilisés en particulier pour la distribution des signaux (si nous ne changeons pas de session, le processus recevra tous les signaux du père -en particulier le signal SIGTERM- et mourra fatalement). Suite à l'appel de cette primitive, le processus appelant devient le leader du nouveau groupe.

```
if (setsid() < 0) { perror("echec setsid"); exit(-1); }
```

Fermeture des flux et des descripteurs de fichier

Il nous faut maintenant fermer tous les descripteurs qui sont devenus inutiles et que l'on a hérités du père.

```
int i;
for (i=getdtablesize()-1; i>=0; i--) { close(i); }
```

Il faut en particulier s'occuper des descripteurs d'entrées et de sorties standards 0, 1 et 2 (stdin, stdout et stderr) qui n'ont pas d'intérêt pour un processus détaché d'un terminal. Si l'on saute cette étape, le processus va par exemple continuer à envoyer la sortie standard vers le terminal de contrôle du père.

Par sécurité, il est préférable d'ouvrir ensuite ces trois descripteurs de fichiers pour les accrocher à un pseudo-terminal.

```
i = open( "/dev/null", "rw" );  /* flux stdin */
dup(i);                         /* flux stdout */
```

```
dup(i);                          /* flux stderr */
```

Changement du répertoire courant

Imaginons que vous lanciez votre démon dans un répertoire où vous conservez des données critiques ; si d'aventure il vous arrive de créer un nouveau fichier dans le répertoire courant (volontairement ou suite à une faille exploitée de l'extérieur), vous pourrez alors compromettre vos données. Il est préférable de changer de répertoire courant par la primitive **chdir()**.

```
if (chdir("/") < 0) { perror("echec chdir"); exit(-1); }
```

De plus, imaginons le cas où votre démon est lancé dans une partition que vous souhaitez démonter ultérieurement, cela ne sera alors pas possible. Il est de toute façon plus que conseillé de changer le répertoire de travail pour la racine ou tout autre répertoire à la convenance de l'application (/tmp par exemple).

Masque de création des fichiers

Pour des raisons évidentes, il arrive parfois qu'un serveur doive s'exécuter sous les droits de root. Il est donc préférable de protéger implicitement les fichiers qui seront éventuellement créés en jouant sur les privilèges avec la primitive **umask()**.

```
umask(0222);
```

Ce masque est utilisé par l'appel-système **open()** pour positionner les permissions d'accès sur les fichiers créés par notre démon. Par exemple, la valeur par défaut 022 permet de supprimer le droits d'exécution sur tous les fichiers : 0777 & ~022 = 0755 (exemple où le mode de création du fichier équivaut à 0777).

Copie unique

La majorité des démons n'autorise le lancement que d'une seule instance à la fois pour des raisons évidentes d'accès concurrentiel aux ressources de la machine : il faut donc s'assurer au lancement du démon que le programme ne tourne pas déjà ; auquel cas, il faudra sortir proprement.

Nous pouvons utiliser pour cela un lock sur un fichier par l'emploi de la primitive **lockf()** en profitant au passage de l'occasion pour mémoriser dans le fichier le numéro d'identifiant du processus (ce qui est généralement une excellent idée afin de faciliter la maintenance du démon). On utilise la constante **F_TLOCK** pour forcer un appel non-bloquant et pour renvoyer une erreur si le fichier est déjà verrouillé.

```
char chaine[32];
int fic = open ( "/var/lock/demon.pid", O_RDWR|O_CREAT, 0640 );
if (fic < 0) { perror("echec open"); exit(-1); }
if (lockf(fic,F_TLOCK,0) < 0) { perror("echec lockf"); exit(-1); }
sprintf( chaine, "%d", getpid() );
write( fic, chaine, strlen(chaine) );
```

Gestion des signaux

Un processus peut recevoir des signaux de la part d'autres processus ou d'un utilisateur : si l'on ne veut pas voir notre démon mourir brusquement et sans raison, il est nécessaire de mettre en place une politique adaptée de traitement de chaque signal. Cela simplifie aussi largement la maintenance.

Le minimum concerne le signal **SIGHUP** (par convention, utilisé pour forcer un démon à relire sa configuration) et le signal **SIGTERM** (pour sortir proprement de l'application).

```
void signal_handler( int signal ) {
    switch( signal ) {
        case 1:         /* SIGHUP */
            /* on force la relecture de la configuration */
            break;
        case 15:        /* SIGTERM */
            /* on libère les ressources du serveur */
            exit(0);
            break;
        }
    }
signal(  1, signal_handler );
signal( 15, signal_handler );
```

Traçabilité des évènements

Un démon s'isole volontairement du reste du système. Il n'est alors pas toujours évident de savoir si tout se passe bien : d'où l'idée de pouvoir tracer des messages d'erreur, des warning ou encore des informations de debug.

Pour cela, il est conseillé de tracer tous les messages dans un fichier de log.

Deux approches sont généralement usitées : tracer les messages dans un fichier réservé strictement à notre démon ou utiliser le démon syslogd qui permet de regrouper des messages systèmes de provenance diverse dans des classes et de rediriger ces classes dans des fichiers (en général, **`/var/log/messages`**).

Cette deuxième solution est plus flexible que la première mais offre le désavantage de noyer ses propres messages avec ceux des autres démons (sauf configuration d'une classe dédiée). Tout est affaire de goût.

Pour tracer vos messages dans un fichier, vous pouvez utiliser le prototype suivant :

```
void trace_message( char *fichier, char *message ) {
    FILE *file;
    file = fopen( fichier, "a" );
    if (file == NULL) { return; }
    fprintf( file, "%s : %s\n", message );
    fclose( file );
    }
trace_message( "/var/log/demon_errors.log", "probleme fichier" );
trace_message( "/var/log/demon_infos.log", "traitement connexion XXX" );
```

Le démon **syslogd** se configure par le fichier **`/etc/syslog.conf`** qui permet de choisir les classes et les règles de redirection des messages. Chaque message possède un niveau de priorité comme par exemple **`LOG_ERR`**, **`LOG_WARNING`** ou **`LOG_INFO`**. La documentation vous aidera à utiliser au mieux ce démon.

On choisit dans cet exemple de tracer l'identifiant de processus de notre démon et de loguer l'évènement dans la classe **`LOG_DAEMON`** (la primitive **`syslog`** se comporte à l'identique de la primitive **`printf`** en ce qui concerne le formatage des données) :

```
openlog( "mon_demon", LOG_PID, LOG_DAEMON );
syslog( LOG_INFO, "traitement connexion %s", mon_client );
syslog( LOG_ERR,  "impossible d'ouvrir le fichier" );
closelog();
```

Emprisonnez votre démon

Mettre en place une prison pour notre démon consiste à limiter la visibilité du système de fichiers par un processus afin de se prévenir contre toute tentative d'altération des données en dehors du périmètre autorisé.

La primitive **`chroot()`** permet justement de mettre en place ce mécanisme : chroot remplace le répertoire racine du processus en cours par un chemin absolu spécifié en argument. Ce répertoire sera utilisé par la suite comme origine des chemins commençant par « `/` ». Ce chemin sera hérité logiquement lors de toute création d'un fils.

```
chroot( "/tmp" );
```

Seul le root peut effectuer un changement de répertoire racine, voire se sortir ultérieurement de la prison avec un peu d'astuce ; il est donc préférable par la suite de changer l'utilisateur du démon pour se prévenir définitivement d'un tel scénario.

Par contre, un petit bémol doit être apporté si vous prévoyez d'utiliser cette commande dans une application nécessitant un niveau élevé de sécurité. Un article très intéressant parut dans **Misc** à fait la démonstration que chroot n'est absolument pas à voir comme la solution ultime pour sécuriser un code : *Misc 9*, « *Chroot () sécurité illusoire ou illusion de sécurité ?* » par *Brad Spengler* (un des auteurs du site grsecurity.net).

Changer d'utilisateur

Un démon n'a pas nécessairement besoin de tourner avec les droits de root ; c'est même fortement déconseillé (sauf cas extrême ne pouvant être résolu par un **sudo** ou un **sticky-bit**). Il est alors préférable de changer l'utilisateur et le groupe par les primitives **`setuid()`** et **`setgid()`**. Attention, une fois ceci effectué, il est impossible au programme de retrouver ses privilèges de root.

```
setuid( 1000 ); /* droits restreints */
```

```
setgid( 100 );
```

Le mot de la fin

Il existe une primitive toute simple qui permet de détacher un simple processus de son terminal de contrôle et l'autorise à s'exécuter en arrière plan à la façon d'un démon système. Cette primitive **daemon()** applique les étapes du **fork()**, du **setsid(),** du **chdir()** pour enfin rediriger l'entrée standard, la sortie standard et la sortie d'erreur vers `/dev/null` ; soit le minimum vital pour transformer à peu de frais un processus standard en démon.

A vous de choisir la méthode à utiliser sachant que vous apprendrez beaucoup en appliquant consciencieusement chacunes des techniques que nous venons d'aborder ; le comment et le pourquoi sont souvent plus importants que le résultat final.