Lancer une application dès le démarrage du système *

Lorsqu'un programme doit démarrer dès l'initialisation du système, sans qu'un utilisateur n'ait besoin de se connecter pour le lancer, il est indispensable de respecter certaines règles pour simplifier la mise en service et éviter les problèmes de sécurité. Nous examinerons dans cet article les principaux points importants à envisager, en détaillant les différents types d'interfaces utilisateur que l'application peut mettre en œuvre et les configurations qui en découlent.

.

Introduction

Le développement de Linux permet de multiplier le nombre de stations de travail de type Unix en employant du matériel peu coûteux, dont les performances et la fiabilité connaissent néanmoins une amélioration constante. De plus, l'engouement du public pour Linux met à la disposition des entreprises un nombre important d'administrateurs système et développeurs compétents (c'est peut-être finalement cet engouement du public qui manquait aux projets BSD libres pour une large diffusion dans le monde professionnel).

On peut ainsi rencontrer de plus en plus fréquemment, surtout dans les environnements de production, des sites composés d'une multitude de machines, chacune étant dédiée à une seule et unique tâche. Telle station contient un système d'enregistrement et de statistiques, telle autre un programme de visualisation temps-réel des données, telle autre enfin un outil de diagnostique des mesures capable d'alerter un opérateur en cas de danger potentiel. Une machine dédiée peut également servir à superviser les autres stations et le réseau lui-même, détectant les pannes, les saturations ou les absence de trafic. Dans d'autres environnements, certains ordinateurs seront réservés pour la journalisation des connexions externes, et d'autres serviront à mettre en lumière les tentatives d'intrusion.

Pour simplifier la tâche de l'administrateur et des utilisateurs, il est souhaitable que l'application principale d'une machine soit lancée automatiquement lors du démarrage de cette dernière, sans avoir besoin de passer par une phase de connexion utilisateur / mot de passe. Il faut quand même observer quelques règles de prudence concernant la sécurité du système pour éviter qu'une maladresse non intentionnelle dégrade la configuration de la station.

Nous supposerons donc que l'on dispose d'une application qui doit être installée pour démarrer dès l'initialisation de la machine. En fonction de l'interface utilisateur la configuration sera différente, aussi examinerons-nous plusieurs cas. Pour des raisons de sécurité nous pourrons aussi être amenés à lancer l'application indirectement par le biais d'un petit programme gérant les identifiants de l'utilisateur.

Initialisation du système

Sans entrer vraiment dans les détails, nous devons parcourir rapidement les diverses étapes du démarrage d'une machine pour bien voir où notre application devra s'insérer.

Lorsqu'un ordinateur est mis sous tension, un programme se trouvant en mémoire morte est exécuté. Il vérifie l'état de la mémoire vive, initialise les ports d'entrée-sortie puis, sous contrôle du paramétrage inscrit dans le setup, charge et lance le code se trouvant sur les premiers secteur du disque dur, d'une disquette, voire d'un CD rom. Sur la plupart des stations Linux il s'agit du programme de lancement Lilo, mais on peut aussi rencontrer une image du noyau directement inscrite sur une disquette. Quoiqu'il en soit, l'initialisation du système d'exploitation commence véritablement à ce moment avec le chargement en mémoire du noyau, et son démarrage. Celui-ci détecte et initialise les périphériques, prépare ses structures de données, puis lance un premier processus, *init*.

Le processus *init*, dont le PID vaut 1 par convention, consulte le fichier **`/etc/inittab`**, et lance les applications qui y sont mentionnées. Un système Linux dispose de plusieurs niveaux d'exécution, qui correspondent à des configurations différentes des applications démarrées. Le niveau d'exécution par défaut est indiqué en début de fichier, par exemple :

```
# Démarrage par défaut au niveau 5
id:5:initdefault:
```

Sur la plupart des distributions Linux, le niveau d'exécution 3 correspond à un démarrage avec tous les utilitaires standards, mais en mode texte (console) alors que les niveaux 4 ou 5 correspondent au démarrage graphique avec X-Window. On rencontre alors une ligne du genre :

```
# Lancer XDM (X-Window Display Manager)
dans les niveaux 4 et 5
x:45:respawn:/usr/X11R6/bin/xdm -nodaemon
```

---

* Cet article a été publié dans le numéro 24 de Linux Magazine France, en Janvier 2001.

Dans le fichier **/etc/inittab** se trouvent également les lancement des démons système (comme *sendmail*, *crond* ou *inetd*) ainsi que les utilitaires de la famille *getty* qui gèrent la connexion sur des consoles virtuelles en mode texte.

Le fichier **/etc/inittab** sera donc le point central de notre configuration, et mérite que l'on s'arrête quelques instants sur sa composition, qui est relativement simple. Les lignes commençant par le caractère **#** représentent des commentaires, qui sont ignorés. Sinon, les autres lignes ont la structure suivante :

```
identifiant:niveaux:type_action:commande
```

L'identifiant doit être unique. Il s'agit d'un ou deux caractères qui seront essentiellement utilisés pour diagnostiquer les problèmes lors de la lecture des fichiers-journaux du système. Le second champ indique la liste des niveaux d'exécution pour lesquels la ligne s'applique. Certaines lignes ont un rôle particulier et laissent ce champ vide, mais cela ne nous concernera pas ici. Le type d'action indiqué en troisième position permet de qualifier la ligne. Les types les plus intéressants pour nous sont :

- **once** : Exécuter la commande se trouvant dans le dernier champ une et une seule fois, lors de l'entrée au niveau d'exécution considéré (au démarrage normalement).

- **respawn** : Lancer la commande indiquée en quatrième position, puis la redémarrer à chaque fois qu'elle se terminera. Il s'agit généralement du comportement le plus intéressant, puisque l'application redémarrera si l'utilisateur l'arrête par erreur, ou si elle s'interrompt pour une raison quelconque.

- **wait** : Lancer la commande lorsqu'on entre dans le niveau d'exécution indiqué (comme avec **once**), mais attendre que le processus se termine avant de passer à la suite. En général cette action est utilisée pour démarrer des démons système qui vont basculer d'eux-mêmes à l'arrière-plan, en rendant la main à init.

D'autres types d'action existent, comme **initdefault** qui spécifie le niveau d'exécution au démarrage, **sysinit** qui paramètre le lancement des applications durant le boot, ou encore **powerfail** ou **powerokwait** qui gèrent les coupures et reprises de courant grâce à un système d'alimentation protégée.

La partie commande de la ligne devra comporter le chemin d'accès complet jusqu'au fichier exécutable à lancer, suivi de ses éventuels arguments.

Interface utilisateur

En fait, la configuration que nous adopterons pour lancer automatiquement notre application au démarrage du système dépendra essentiellement de son interface utilisateur, qui nous indiquera où et comment placer notre programme. Nous allons étudier en premier lieu le cas des applications n'ayant pas de contact avec l'utilisateur. Nous examinerons ensuite les situations où le programme dispose d'une interface en mode texte, puis en mode graphique.

Application sans interaction avec l'utilisateur

Le cas le plus courant d'application lancée directement au démarrage du système est celui des démons. Il s'agit de programme s'exécutant à l'arrière-plan, sans terminal de contrôle, et chargés pour la plupart de travaux administratifs comme l'exécution de jobs différés (*cron*), le routage et la distribution du courrier électronique (*sendmail*), la gestion des connexions réseau (*inetd*) ou l'impression en tâche de fond (*lpd*).

Un tel programme doit respecter un certain nombre de règles :

- basculer à l'arrière-plan, même s'il a été lancé au premier-plan par le shell ;

- créer une nouvelle session de processus pour ne pas avoir de terminal de contrôle ;

- fermer tous les descripteurs de fichiers dont il peut avoir hérité de son processus père ;

- revenir à la racine du système de fichiers pour ne pas risquer de bloquer une partition que l'administrateur ne pourrait pas démonter.

Ces opérations sont réalisées dès l'initialisation du programme par quelques lignes de code. Par exemple le morceaux de programme suivant lance sous forme de démon l'application proprement dite, représentée par la routine **mon_application()** :

```
  /* demon.c */

  #include <limits.h>
  #include <unistd.h>

  int
mon_application (int argc, char * argv []);

    int
  main (int argc, char * argv [])
  {
    int fd;
    if (fork () != 0)
      exit (0);
    setsid();
    for (fd = 0; fd < OPEN_MAX; fd ++)
      close (fd);
    chdir ("/");

    return (mon_application (argc, argv));
  }
```

Démarrage du logiciel

Pour lancer l'exécution du programme, nous allons agir dans **/etc/inittab**. Il s'agit de la méthode la plus générale, fonctionnant sur toutes les distributions de Linux, et même sur la majeure partie des Unix. Certains administrateurs rechignent à modifier directement ce fichier, en préférant agir sur les scripts qu'il lance automatiquement. On peut ainsi ajouter les commandes que nous voulons dans le script **rc.local** par exemple. C'est une bonne méthode car elle évite de surcharger un fichier système en y apportant des modifications à chaque fois qu'un démon est ajouté. Toutefois nous nous plaçons ici dans le cas d'une machine dédiée essentiellement à une application donnée, et l'intervention sur le fichier **/etc/inittab** se justifie très bien si elle est limitée à l'installation de ce logiciel.

L'application que nous lançons passe automatiquement à l'arrière-plan. La commande utilisée pour le démarrage va donc se terminer presque immédiatement (dans le **exit(0)** suivant le **fork()**). Il n'est donc pas question de demander un démarrage de type **respawn**, car *init* essayerait de la relancer tout de suite, bouclant ainsi sur cette ligne pendant quelques secondes au bout desquelles il abandonnerait en différant le lancement pour réessayer au bout de quelques minutes. En attendant, la boucle **respawn** aurait lancé un nombre important de démons en parallèle, surchargeant le système.

Pour démarrer un démon, on préfère en général un lancement de type **wait**, *init* attendant que le processus ait basculé à l'arrière-plan pour continuer son travail. On ajoute donc dans **/etc/inittab** une ligne du type :

```
lm:3:wait:/usr/local/bin/mon_demon
```

Les caractères du premier champ sont choisis arbitrairement de manière à former un identificateur unique dans le fichier. Le niveau d'exécution indiqué ensuite est celui auquel le démon sera actif. Sauf exception, il s'agira du niveau d'exécution indiqué dans la ligne **initdefault** au début du fichier.

Transmission de messages administratifs

L'application n'ayant pas de possibilité de dialoguer avec l'utilisateur par l'intermédiaire du terminal, elle doit implémenter son propre mécanisme pour interagir avec son environnement. Selon les cas, elle peut employer des sockets, des tubes nommés, des fichiers et des signaux, tout cela au gré du programmeur ou plutôt du concepteur du logiciel. En revanche, pour pouvoir transmettre des informations de diagnostique, indispensables pour la mise au point et le débogage du système, il faut disposer d'un mécanisme remplaçant l'écriture de message sur la sortie d'erreur.

Nous employons pour cela les possibilités offertes par le démon syslogd. Il s'agit d'un système de journalisation qui centralise tous les messages d'erreur ou de diagnostique émis par les applications l'invoquant, et qui permet d'enregistrer dans un fichier, d'afficher à l'écran, d'envoyer par courrier électronique, ou d'imprimer les traces d'exécution. La configuration du démon *syslogd* lui-même se paramètre à l'aide du fichier **/etc/syslog.conf** comme cela a été décrit dans l'article « *syslog : maîtrisez l'historique* » de Vincent Renardias, paru dans le numéro 19 de Linux Magazine.

Pour le programmeur, l'utilisation de *syslog* est très simple, puisqu'elle se réduit à l'invocation de trois routines de la bibliothèque C, déclarées dans **<syslog.h>**. Ces fonctions sont les suivantes :

- **void openlog (char *ident, int option, int type)** : cet appel facultatif au début du programme initialise le fonctionnement des routines. On transmettra en premier argument le nom du programme, ce qui permettra de l'identifier dans les fichiers de journalisation, en second argument la constante symbolique **LOG_PID** qui permettra l'ajout systématique du numéro de processus dans les traces d'exécution. En dernier argument on identifiera le type d'application avec la constante **LOG_USER**. On trouvera plus de détail dans la page de manuel **openlog(3)**. Si on n'invoque pas **openlog()**, une initialisation par défaut aura lieu lors de la première journalisation.

- **void closelog (void)** : appelée facultativement en fin de programme, cette routine ferme la communication avec le démon *syslogd*.

- **void syslog (int priorite, char * format...)** : cette routine fonctionne un peu comme **printf()**, en utilisant une chaîne de format pour présenter des données. Le premier argument représente l'urgence du message. Les valeurs les plus courantes sont **LOG_ERR** en cas d'erreur grave (par exemple l'application risque de s'arrêter), **LOG_INFO** pour transmettre un message significatif mais sans danger (par exemple une connexion vient d'être établie), et **LOG_DEBUG** pour une information utile uniquement en débogage (par exemple allocation de 1024 octets).

La configuration permet d'orienter les messages à sa guise en fonction de leur provenance et de leur urgence. Aussi n'hésiterons-nous pas à transmettre des avertissements pour toutes les situations significatives. Même en fonctionnement normal, il est très commode pour l'administrateur de s'assurer de la bonne santé d'un système fonctionnant de manière autonome en jetant un coup d'œil dans les fichiers de *logs* comme **/var/log/messages**.

L'utilisation du système *syslog* n'est pas réservée aux applications en langage C. Il existe un utilitaire nommé **logger** qui offre une interface pour les scripts shell. On l'utilisera souvent avec les options :

- **-t nom** : indiquant le nom du programme, afin de faciliter l'exploration des fichiers de journalisation ;

- **-i** : qui réclame l'inscription du PID dans les lignes de message ;

- **`-p priorité`** : la priorité est indiquée sous une forme particulière : **`type.urgence`**. Le type sera généralement **`user`** et l'urgence sera **`err`**, **`info`** ou **`debug`**.

Il est ensuite suivi du message proprement dit. Voici un exemple :

```
  #! /bin/sh
  # log.sh exemple d'invocation de syslog

  logger -i -t $0 -p user.err "Erreur grave"

  logger -i -t $0 -p user.info "Information
utile"

  logger -i -t $0 -p user.debug "Message de
déboguage"
```

Sur la plupart des systèmes les messages de déboguage ne sont pas enregistrés. Nous allons donc modifier notre configuration dans **`/etc/syslog.conf`** pour les diriger vers un nouveau fichier, et demander à *syslogd* de relire sa configuration en lui envoyant un signal *SIGHUP*. La ligne ajoutée dans **`/etc/syslog.conf`** est :

```
  user.=debug    /var/log/debug
```

Voici un exemple d'utilisation de notre script :

```
# killall -HUP syslogd
# ./log.sh
# tail /var/log/messages
  [...]
Nov 13 16:34:15 venux syslogd 1.3-3: restart.
nov 13 16:34:29 venux ./log.sh[2859]: Erreur
grave
nov 13 16:34:29 venux ./log.sh[2860]:
Information utile
# tail /var/log/debug
nov 13 16:34:29 venux ./log.sh[2861]: Message
de déboguage
#
```

Nous avons donc vu comment un programme sans interface utilisateur peut transmettre quand même des informations administratives au système. Ce mécanisme est aussi largement utilisé dans des programmes interactifs, quand un message doit être dirigé vers l'administrateur système et non pas vers l'utilisateur final.

Limitation des privilèges

Un programme lancé par le processus *init* s'exécute automatiquement sous l'identité *root*. Pour un démon tournant à l'arrière-plan, et dont les communications avec les utilisateurs sont soigneusement contrôlées, cela ne pose pas de gros problème. Toutefois lorsque l'application peut être manipulée par un utilisateur quelconque, comme nous le rencontrerons dans les logiciels étudiés plus loin, un véritable danger se présente.

Sans même mettre en doute les intentions bienveillantes de l'utilisateur, une simple maladresse sous l'identité *root* peut être catastrophique ; imaginez par exemple que l'on sauvegarde les statistiques du mois en cours dans **`/etc/passwd`**, **`/vmlinuz`**, ou **`/dev/hda`**.

Il faut donc arriver à restreindre les privilèges de l'application. Pour cela on peut dans certains cas utiliser les bits *Set-UID* et *Set-GID* du fichier exécutable afin que le programme fonctionne automatiquement sous l'identité du propriétaire et du groupe du fichier (que l'on donnera à un utilisateur *lambda* sans privilège). Ce mécanisme peut parfois être suffisant, mais deux inconvénients se présentent quand même :

- Le noyau n'honore pas les bits *Set-UID* et *Set-GID* sur les fichiers scripts pour des raisons de sécurité. Naturellement, cette restriction est en principe destinée à protéger le système dans le cas contraire, où les bits *Set-UID* et *Set-GID* augmentent les privilèges du processus. Malheureusement la limitation est également valable dans notre situation, où l'on essaye de diminuer le niveau de privilèges. Nous ne pourrons donc pas utiliser ce moyen pour traiter les fichiers scripts. Notez que cela concerne pratiquement tous les scripts (Perl faisant exception grâce à un artifice basé sur un programme binaire *Set-UID*) et pas seulement le shell.

- Lorsqu'un programme exécutable a son bit *Set-UID* à 1, il disposera d'un UID effectif correspondant au propriétaire du fichier. Cet UID effectif sera employé pour toutes les vérifications d'autorisation. Mais il continuera à disposer d'un UID réel correspondant à celui de l'utilisateur qui l'a lancé, ceci à des fins d'identification des actions. Un fichier *Set-UID lambda* démarré par *root* aura donc un UID effectif *lambda* et un UID réel *root*. Les vérifications d'accès aux fichiers du système se faisant avec l'UID effectif, les privilèges sont bien descendus. Mais un autre problème se pose : le processus est capable à tout moment de reprendre en UID effectif la valeur de son UID réel (*root*). Ce qui signifie que le programme va devenir une cible de choix pour les attaques de sécurité du type débordements de buffers.

Il nous faut donc disposer d'un moyen de diminuer les privilèges des deux types d'UID, et de lancer des scripts avec ces mêmes modifications.

Nous allons écrire un programme qui prend en argument une chaîne de caractères correspondant à un fichier exécutable suivi d'éventuels arguments. Ce processus va modifier ses UID et GID effectifs et réels pour correspondre à ceux du fichier exécutable visé, puis va l'invoquer avec la commande **`system()`**.

Ce programme ne dispose pas personnellement de privilèges particuliers, car il n'est conçu que pour en perdre. Ainsi il n'a d'intérêt que lorsqu'il est invoqué par *root* pour faire démarrer un programme appartenant à un utilisateur *lambda*. Comme l'invocation sera figée dans des fichiers système sous le contrôle unique de l'administrateur, ce programme ne peut en aucun cas servir à exploiter des failles de sécurité. On peut donc sans risque utiliser la fonction **`system()`** si dangereuse dans d'autres circonstances (programmes Set-UID *root* par exemple).

```
/* lanceur.c */
#include <stdlib.h>
```

```
    #include <string.h>
    #include <syslog.h>
    #include <unistd.h>
    #include <sys/stat.h>

    void
set_uid (const char * fichier)
{
    struct stat etat;
    char *      executable;
    char *      blanc;
    /* Copie du nom du fichier pour pouvoir
le modifier */
    if ((executable = (char *) malloc (strlen
(fichier) + 1)) == NULL) {
        syslog (LOG_ERR, "Mémoire
manquante");
        exit (1);
    }
    strcpy (executable, fichier);
    /* Remplacement du premier blanc ou
tabulation par un \0 */
    /* Ceci permet de supprimer les éventuels
arguments */
    blanc = strchr (executable, ' ');
    if (blanc != NULL)
        blanc [0] = '\0';
    blanc = strchr (executable, '\t');
    if (blanc != NULL)
        blanc [0] = '\0';
    /* Lecture des informations concernant le
fichier */
    if (stat (executable, & etat) != 0) {
        free (executable);
        syslog (LOG_ERR, "Fichier %s
inaccessible", executable);
        exit (1);
    }
    free (executable);
    /* On fixe les UID/GID effectifs et réels
*/
    setreuid (etat . st_uid, etat . st_uid);
    setregid (etat . st_gid, etat . st_gid);
}

    int
main (int argc, char * argv [])
{
    openlog (argv [0], 0, LOG_USER);

    if (argc != 2) {
        syslog (LOG_ERR, "Syntaxe : %s
commande", argv [0]);
        exit (1);
    }

    /* Utilisation des UID et GID du fichier
exécutable */
    set_uid (argv [1]);

    /* Lancement du programme */
    system (argv [1]);
    return (0);
}
```

L'exécution suivante montre bien que lorsque *root* demande le lancement d'un fichier exécutable appartenant à un autre utilisateur, les UID et GID sont bien modifiés de telle manière que le retour à l'UID *root* nécessite la saisie du mot de passe (ce qui n'aurait pas été le cas si les UID/GID réels n'étaient pas modifiés).

```
  $ cc lanceur.c -o lanceur -Wall
  $ chmod +s lanceur
  $ ls -l lanceur
  -rwsrwsr-x  1 ccb  ccb   15680 Nov 16 11:21
lanceur
  $ cp /bin/sh .
  $ ls -l sh
  -rwxr-xr-x  1 ccb  ccb  373176 Nov 16 11:21
sh
  $ su
  Password:
  # ./lanceur ./sh
  bash$ whoami
  ccb
  bash$ su
  Password:
  su: incorrect password
  bash$ exit
  # exit
  $
```

Notez que si l'on doit transmettre des arguments à la commande lancée, il faut encadrer le tout par des guillemets ainsi :

```
  $ lanceur "/usr/local/bin/mon_application
et des arguments"
```

La ligne de lancement dans le fichier **/etc/inittab** sera donc finalement modifiée ainsi :

```
  lm:3:wait:/usr/local/bin/lanceur
"/usr/local/bin/mon_demon arg1 arg2"
```

Application avec une interface en mode texte

Le cas d'une application disposant d'une interface utilisateur en mode texte est intéressant car nous allons devoir gérer nous-même l'affection d'un terminal virtuel. Nous supposons que le but du système est de faire tourner automatiquement un logiciel employant une interface sur un terminal texte, soit avec les flux d'entrées-sorties standards, soit en employant une bibliothèque plus élaborée comme *ncurses*. Nous voici donc pratiquement revenu au temps du bon vieux Dos. Cette piste n'est d'ailleurs pas totalement incongrue, et pour faire tourner une application ne nécessitant pas la mise en oeuvre de toute la puissance d'un système Unix ni de toutes les possibilités d'interface réseau, le projet *Freedos* peut être très intéressant en permettant à une application de fonctionner à partir d'une disquette (donc sur des systèmes embarqués sans disque dur).

Notre application va devoir s'attribuer un terminal texte à la manière des utilitaires de la famille *getty* (*mingetty*, *mgetty*, *agetty*, etc.) Cela s'obtient en ouvrant un pseudo-fichier **/dev/ttyX**, puis en dupliquant le descripteur obtenu pour l'affecter aux entrées et sorties standards. Le code nécessaire est le suivant :

```
    void
get_tty (const char * tty)
{
    int    fd;
    /* Ouverture du fichier indiqué */
    if ((fd = open (tty, O_RDWR, 0)) <0) {
        syslog (LOG_ERR, "Impossible d'ouvrir
%s", tty);
        exit (1);
```

```
    }
    /* Vérification qu'il s'agit bien d'un
terminal */
    if (! isatty (fd)) {
        syslog (LOG_ERR, "%s n'est pas un
terminal", tty);
        exit (1);
    }
    /* Duplication du descripteur sur stdin,
stdout, et stderr */
    if ((dup2 (fd, 0) ><0) || (dup2 (fd, 1)
><0) || (dup2 (fd, 2) ><0)) {
        syslog (LOG_ERR, "Erreur dans
dup2()");
        exit (1);
    }
    close (fd);
}
```

Cette routine est appelée en lui passant en argument le nom du pseudo-fichier **/dev/ttyX** désiré. Pour que l'ouverture réussisse, il faut que le programme s'exécute sous l'identité root. Le plus simple est donc de réaliser l'ouverture avant de modifier les UID et GID comme nous l'avons fait plus haut. Le programme de lancement est alors modifié pour recevoir en premier argument le nom du terminal à ouvrir.

```
    /* lanceur_texte.c */
    #include <fcntl.h>
    #include <stdlib.h>
    #include <string.h>
    #include <syslog.h>
    #include <unistd.h>
    #include <sys/stat.h>

    void
set_uid (const char * fichier)
{
    /* Même code que dans lanceur.c */
}

    void
get_tty (const char * tty)
{
    /* Code présenté ci-dessus */
}

    int
main (int argc, char * argv [])
{
    openlog (argv [0], 0, LOG_USER);

    if (argc != 3) {
        syslog (LOG_ERR, "Syntaxe : %s tty
fichier", argv [0]);
        exit (1);
    }
    /* Ouverture de la console indiquée en
premier argument */
    get_tty (argv [1]);

    /* Utilisation des UID et GID du fichier
exécutable */
    set_uid (argv [2]);

    /* Lancement du programme */
    system (argv [2]);
    return (0);
}
```

Le démarrage du programme va se faire en principe sur la première console virtuelle, **/dev/tty1**, avec le niveau d'exécution 3 par défaut. Il va donc falloir remplacer le lancement de *getty* sur cette console. Il est probable que dans votre fichier /etc/inittab vous rencontriez une ligne du type

```
1:2345:respawn:/sbin/mingetty tty1
```

ou

```
1:2345:respawn:/sbin/mgetty -rb tty1
```

Il va falloir supprimer cette ligne, et la remplacer par:

```
1:3:respawn:/usr/local/bin/lanceur_texte
/dev/tty1 "/usr/local/bin/mon_appli"
```

Le niveau d'exécution sera adapté à celui par défaut. Ici l'application ne passe pas en arrière-plan, contrairement aux démons que nous avons vus plus haut, aussi peut-on employer un lancement de type **respawn** qui redémarrera le logiciel si l'utilisateur l'arrête par erreur. On n'oubliera pas de donner la propriété de l'exécutable **mon_appli** à un utilisateur et un groupe non privilégiés.

Le lecteur désireux d'expérimenter rapidement cette configuration pourra utiliser comme application en mode texte le *Midnight Commander* "**mc**" qui utilise la bibliothèque *ncurses* :

```
# cp /usr/bin/mc /usr/local/bin/
# chown nobody.nobody /usr/local/bin/mc
#
```

Éditer le fichier **/etc/inittab**, mettre en commentaire la ligne contenant le *getty*, et en ajouter une faisant démarrer le lanceur :

```
# Run gettys in standard runlevels
# 1:2345:respawn:/sbin/mingetty tty1
1:2345:respawn:/usr/local/bin/lanceur_texte
/dev/tty1 /usr/local/bin/mc
```

Demander à *init* de relire ce fichier.

```
# kill -1 1
#
```

Basculer sur le terminal virtuel 1 (*Contrôle-Alt-F1*), et terminer le *getty* qui y fonctionne (avec *Contrôle-D* par exemple). À partir de ce moment, **mc** démarrera automatiquement sur cette console, avec une identité sans privilège (ce que l'on peut vérifier en essayant de déplacer des fichiers système).

Application graphique sous X-Window

Dans la majeure partie des cas actuels on désirera probablement faire tourner une application graphique sous X-Window plutôt qu'un logiciel en mode texte. Les problèmes qui se posent alors sont d'un autre ordre. Nous allons présenter ici une méthode pour les résoudre, mais d'autres techniques seraient sans doutes applicables, chacun pouvant essayer d'expérimenter à son gré.

Le principe retenu ici est de regrouper toutes les commandes nécessaires dans un seul script shell que l'on lance directement dans **/etc/inittab**. On supprimera alors la ligne qui démarre *Xdm*, *Kdm*, *Prefdm* ou autres :

```
x:45:respawn:/usr/X11R6/bin/xdm -nodaemon
```

On remplacera cette ligne par :

```
x:45:respawn:/usr/local/bin/graphique.sh
```

Le script shell **graphique.sh** va donc faire démarrer un serveur X-Window en lançant le logiciel **/usr/X11R6/bin/X**. Dès lors l'écran sera initialisé en mode graphique, avec un fond contenant une sorte de damier - assez laid - et un curseur de souris en forme de croix. On peut observer cette configuration en invoquant la commande "**xsetroot -def**" dans une fenêtre Xterm sous X11.

Pour pouvoir lancer notre application, nous devons reprendre la main, aussi le lancement de X doit-il être suivi d'un & pour le passer à l'arrière-plan. Nous pouvons dorénavant exécuter notre logiciel, en l'invoquant au travers du programme lanceur vu plus haut, qui limite les privilèges en modifiant les UID et GID. Il est obligatoire de faire suivre l'invocation du programme de l'argument "**-display :0.0**" qui sera analysé par la bibliothèque Xt, pour que l'application entre en contact avec le bon serveur X. Pour nos expériences, je propose de recopier le fichier exécutable **xterm** et de modifier son appartenance :

```
# cp /usr/X11R6/bin/xterm /usr/local/bin/
# chown nobody.nobody /usr/local/bin/xterm
#
```

Notre premier script **graphique.sh** aura donc cette allure :

```
#! /bin/sh
/usr/X11R6/bin/X &
/usr/local/bin/lanceur "/usr/local/bin/xterm
-display :0.0"
```

Cela fonctionne, mais l'ensemble est un peu décevant, le fond d'écran présente encore son allure de damier fatiguant pour les yeux, et le curseur de la souris conserve sa forme de croix X11. Nous voudrions donc modifier ces paramètres pour rendre l'aspect plus attrayant. Pour cela, il suffit d'invoquer l'utilitaire *xsetroot*, dont l'option **-solid** permet de préciser une couleur unie pour le fond, et l'option **-cursor_name** autorise le chargement d'un curseur de souris différent. Les noms des différents curseurs sont mentionnés dans le fichier **/usr/include/X11/cursorfont.h** (avec un préfixe **XC_** à supprimer). Le curseur le plus courant est *left_ptr* qui affiche une petite flèche dirigée vers le haut à gauche.

Nous essayons donc d'invoquer *xsetroot* avant notre application, et... cela ne fonctionne pas ! En effet dès que la connexion entre le serveur X et son premier client (*xsetroot*) est rompue, le serveur se termine. Il faut donc que notre application démarre en premier, à l'arrière-plan, et que nous invoquions *xsetroot* par la suite. On peut imaginer quelque chose comme :

```
#! /bin/sh
/usr/X11R6/bin/X &
/usr/local/bin/lanceur "/usr/local/bin/xterm
-display :0.0" &
/usr/X11R6/bin/xsetroot -solid black -
cursor_name left_ptr -display :0.0
```

Le problème qui apparaît à présent est que ce script se termine presque tout de suite puisque X et xterm sont à l'arrière-plan et que *xsetroot* dure très peu de temps. Comme nous avons configuré notre fichier **/etc/inittab** avec une commande **respawn**, *init* essaye de relancer le script immédiatement, et finit par boucler.

La solution est de faire appel aux possibilités offertes par Bash pour passer un processus à l'arrière-plan, mémoriser son PID, et attendre la fin d'un processus particulier. Le script final va donc avoir l'aspect suivant :

```
#!/bin/sh

# Lancer X à l'arrière-plan
/usr/X11R6/bin/X &
# Mémoriser le PID de la dernière commande en
arrière-plan (X)
PID_X=$!

# Lancer l'application à l'arrière-plan
/usr/local/bin/lanceur "/usr/local/bin/xterm
-geometry +200+200 -display :0.0 " &
# Mémoriser son PID
PID_APPLI=$!

# Configurer le fond d'écran et le curseur
avec xsetroot
/usr/X11R6/bin/xsetroot -solid black -
cursor_name left_ptr -display :0.0

# Attendre la fin de l'application
wait $PID_APPLI

# Tuer par précaution le serveur X
kill -TERM $PID_X
```

Cette fois-ci le résultat est exactement ce que l'on attendait. Le fond d'écran est acceptable, et l'application est relancée - ainsi que le serveur X - si jamais elle se termine. Le fait qu'aucun gestionnaire de fenêtre ne soit lancé prive l'application des bordures, barre de titre, etc. mais cela n'est pas contraignant si le poste est dédié à une seule utilisation donc a fortiori à une seule fenêtre principale.

Conclusion

Nous avons étudié trois manière de mettre en place une application pour qu'elle soit automatiquement lancée au démarrage de la machine, sans nécessiter d'intervention de l'utilisateur. La configuration d'une application sans interface utilisateur se fait de manière assez classique, en employant les techniques usuelles des démons Unix.

En revanche la mise en service d'applications s'appuyant sur des interfaces en mode texte ou graphique est plus complexe, mais le résultat est intéressant car il montre bien la souplesse de paramétrage d'une machine Unix. J'ai utilisé avec succès ces configurations dans des logiciels de supervision de communication série (en mode texte), et des systèmes de visualisation graphique destinés à du personnel peu familiarisé avec Unix et X-Window (pompiers).

Christophe Blaess

<ccb@club-internet.fr>

http://perso.club-internet.fr/ccb/