4

Exécution des programmes

Ce chapitre va être principalement consacré aux débuts d'un processus. Tout d'abord, nous examinerons les méthodes utilisables pour lancer un nouveau programme, ainsi que les mécanismes sous-jacents, qui peuvent conduire à un échec du démarrage.

Nous nous intéresserons ensuite à des fonctions simplifiées, permettant d'utiliser une application indépendante comme une sous-routine de notre logiciel.

Lancement d'un nouveau programme

Nous avons déjà vu que le seul moyen de créer un nouveau processus dans le système est d'invoquer `fork()`, qui duplique le processus appelant. De même, la seule façon d'exécuter un nouveau programme est d'appeler l'une des fonctions de la famille **`exec()`**. Nous verrons également qu'il existe les fonctions `popen()` et `system()`, qui permettent d'exécuter une autre application mais en s'appuyant sur `fork()` et `exec()`.

L'appel de l'une des fonctions `exec()` permet de remplacer l'espace mémoire du processus appelant par le code et les données de la nouvelle application. Ces fonctions ne reviennent qu'en cas d'erreur, sinon le processus appelant est entièrement remplacé.

On parle couramment de l'appel-système `exec()` sous forme générique, mais en fait il n'existe aucune routine ayant ce nom. Simplement, il y a six variantes nommées `execl()`, `execle()`, `execlp()`, `execv()`, `execve()` et `execvp()`. Ces fonctions permettent de lancer une application. Les différences portent sur la manière de transmettre les arguments et l'environnement, et sur la méthode pour accéder au programme à lancer. Il n'existe sous Linux qu'un seul véritable appel-système dans cette famille de fonctions : `execve()`. Les autres fonctions sont implémentées dans la bibliothèque C à partir de cet appel-système.

Les fonctions dont le suffixe commencent par un "`l`" utilisent une liste d'arguments à transmettre de nombre variable, tandis que celles qui débutent par un "`v`" emploient un tableau à la manière du vecteur `argv []`.

Les fonctions se terminant par un "e" transmettent l'environnement dans un tableau envp [] explicitement passé dans les arguments de la fonction, alors que les autres utilisent la variable globale environ.

Les fonctions se finissant par un "p" utilisent la variable d'environnement PATH pour rechercher le répertoire dans lequel se situe l'application à lancer, alors que les autres nécessitent un chemin d'accès complet. La variable PATH est déclarée dans l'environnement comme étant une liste de répertoires séparés par des deux-points. On utilise typiquement une affectation du genre :

```
PATH=/usr/bin:/bin:/usr/X11R6/bin/:/usr/local/bin:/usr/sbin:/sbin
```

Il est préférable de placer en tête de PATH les répertoires dans lesquels se trouvent les applications les plus utilisées afin d'accélérer la recherche. Certains ajoutent à leur PATH un répertoire simplement composé d'un point, représentant le répertoire en cours. Cela peut entraîner une faille de sécurité, surtout si ce répertoire « . » n'est pas placé en dernier dans l'ordre de recherche. Il vaut mieux ne pas le mettre dans le PATH et utiliser explicitement une commande :

```
$ ./mon_prog
```

pour lancer une application qui se trouve dans le répertoire courant.

Quand execlp() ou execvp() rencontrent, lors de leur parcours des répertoires du PATH, un fichier exécutable du nom attendu, ils tentent de le charger. S'il ne s'agit pas d'un fichier binaire mais d'un fichier de texte commençant par une ligne du type :

```
#! /bin/interpreteur
```

le programme indiqué (interpreteur) est chargé, et le fichier lui est transmis sur son entrée standard. Il s'agit souvent de /bin/sh, qui permet de lancer des scripts shell, mais on peut trouver d'autres fichiers à interpréter (/bin/awk, /usr/bin/perl, /usr/bin/wish...). Nous verrons une invocation de script shell plus loin.

Si l'appel exec() réussit, il ne revient pas, sinon il renvoie –1, et errno contient un code expliquant les raisons de l'échec. Celles-ci sont détaillées dans la page de manuel execve(2).

Le prototype de **execve()** est le suivant :

```
int execve (const char * appli, const char * argv [],
            const char * envp []);
```

La chaîne "appli" doit contenir le chemin d'accès au programme à lancer à partir du répertoire de travail en cours ou à partir de la racine du système de fichiers s'il commence par un slash "/".

Le tableau argv[] contient des chaînes de caractères correspondant aux arguments qu'on trouve habituellement sur la ligne de commande.

La première chaîne argv[0] doit contenir le nom de l'application à lancer (sans chemin d'accès). Ceci peut parfois être utilisé pour des applications qui modifient leur comportement en fonction du nom sous lequel elles sont invoquées. Par exemple, /bin/gzip sert à compresser des fichiers. Il est également utilisé pour décompresser des fichiers si on lui transmet l'option -d ou si on l'invoque sous le nom gunzip. Pour ce faire, il analyse argv[0]. Dans la plupart des distributions Linux, il existe d'ailleurs un lien physique nommé /bin/gunzip sur le même fichier que /bin/gzip.

Le troisième argument est un tableau de chaînes déclarant les variables d'environnement. On peut éventuellement utiliser la variable externe globale `environ` si on désire transmettre le même environnement au programme à lancer. Dans la majorité des applications, il est toutefois important de mettre en place un environnement cohérent, grâce aux fonctions que nous avons étudiées dans le chapitre 3. Ceci est particulièrement nécessaire dans les applications susceptibles d'être installées Set-UID *root*.

Les tableaux `argv []` et `envp []` doivent se terminer par des pointeurs `NULL`.

Pour montrer l'utilisation de `execve()`, nous allons invoquer le shell, en lui passant la commande `echo $SHLVL`. Le shell nous affichera alors la valeur de cette variable d'environnement. *bash* comme *tcsh* indiquent dans cette variable le nombre d'invocations successives du shell qui sont « empilées ». Voici un exemple sous *bash* :

```
$ echo $SHLVL
1
$ sh
$ echo $SHLVL
2
$ sh
$ echo $SHLVL
3
$ exit
$ echo $SHLVL
2
$ exit
$ echo $SHLVL
1
$
```

Notre programme exécutera donc simplement cette commande en lui transmettant son propre environnement. On notera que la commande `echo $SHLVL` doit être transmise en un seul argument, comme on le ferait sur la ligne de commande :

```
$ sh -c "echo $SHLVL"
2
$
```

(L'option `-c` demande au shell d'exécuter l'argument suivant, puis de se terminer.)

exemple_execve.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    extern char ** environ;

    int
main (void)
{
    char * argv[] = {"sh", "-c", "echo $SHLVL", (char *) NULL };

    fprintf(stdout, "Je lance /bin/sh -c \"echo $SHLVL\" :\n");
```

```
    execve("/bin/sh", argv, environ);

    fprintf(stdout, "Raté : erreur = %d\n", errno);
    return 0;
}
```

Voici un exemple d'exécution sous *bash* :

```
$ echo $SHLVL
1
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$ sh
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
3
$ exit
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$
```

Bien entendu, le programme ayant lancé un nouveau shell pour exécuter la commande, le niveau d'imbrication est incrémenté par rapport à la variable d'environnement, consultée directement avec `echo $SHLVL`.

La fonction **`execv()`** dispose du prototype suivant :

```
int execv (const char * application, const char * argv []);
```

Elle fonctionne comme `execve()`, mais l'environnement est directement transmis par l'intermédiaire de la variable externe `environ`, sans avoir besoin d'être passé explicitement en argument durant l'appel.

La fonction **`execvp()`** utilise un prototype semblable à celui de `execv()`, mais elle se sert de la variable d'environnement `PATH` pour rechercher l'application. Nous allons en voir un exemple, qui exécute simplement la commande `ls`.

exemple_execvp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    char * argv[] = { "ls", "-l", "-n", (char *) NULL };

    execvp("ls", argv);

    fprintf(stderr, "Erreur %d\n", errno);
    return 1;
}
```

Lorsqu'on exécute cette application, celle-ci recherche `ls` dans les répertoires de la variable d'environnement `PATH`. Ainsi, en modifiant cette variable pour éliminer le répertoire contenant `ls`, `execvp()` échoue.

```
$ echo $PATH
/usr/bin:/bin:/usr/X11R6/bin:/usr/local/bin:/usr/sbin
$ which ls
/bin/ls
$ ./exemple_execvp
total 12
-rwxrwxr-x   1 500      500          4607 Aug  7 14:53 exemple_execve
-rw-rw-r--   1 500      500           351 Aug  7 14:51 exemple_execve.c
-rwxrwxr-x   1 500      500          4487 Aug  7 15:20 exemple_execvp
-rw-rw-r--   1 500      500           229 Aug  7 15:20 exemple_execvp.c
$ export PATH=/usr/bin
$ ./exemple_execvp
Erreur 2
$ export PATH=$PATH:/bin
$ ./exemple_execvp
total 12
-rwxrwxr-x   1 500      500          4607 Aug  7 14:53 exemple_execve
-rw-rw-r--   1 500      500           351 Aug  7 14:51 exemple_execve.c
-rwxrwxr-x   1 500      500          4487 Aug  7 15:20 exemple_execvp
-rw-rw-r--   1 500      500           229 Aug  7 15:20 exemple_execvp.c
$
```

La fonction `execlp()` permet de lancer une application qui sera recherchée dans les répertoires mentionnés dans la variable d'environnement `PATH`, en fournissant les arguments sous la forme d'une liste variable terminée par un pointeur `NULL`. Le prototype de **`execlp()`** est le suivant :

```
int execlp (const char * application, const char * arg, ...);
```

Cette présentation est plus facile à utiliser que `execvp()` lorsqu'on a un nombre précis d'arguments connus à l'avance. Si les arguments à transmettre sont définis dynamiquement durant le déroulement du programme, il est plus simple d'utiliser un tableau comme avec `execvp()`.

Voici un exemple de programme qui se rappelle lui-même en incrémentant un compteur transmis en argument. Il utilise `argv[0]` pour connaître son nom ; l'argument `argv[1]` contient alors le compteur qu'on incrémente jusqu'à 5 au maximum avant de relancer le même programme.

exemple_execlp.c

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (int argc, char * argv [])
{
    char compteur[2];
    int i;
```

```
    i = 0;
    if (argc == 2)
        sscanf(argv[1], "%d", & i);

    if (i < 5) {
        i ++;
        sprintf(compteur, "%d", i);

        fprintf(stdout, "execlp(%s, %s, %s, NULL)\n",
                        argv[0], argv[0], compteur);
        execlp(argv[0], argv[0], compteur, (char *) NULL);
    }
    return 0;
}
```

```
$ ./exemple_execlp
execlp(./exemple_execlp, ./exemple_execlp, 1, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 2, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 3, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 4, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 5, NULL)
$
```

La fonction **`execl()`** est identique à `execlp()`, mais il faut indiquer le chemin d'accès complet, sans recherche dans `PATH`. La fonction **`execle()`** utilise le prototype suivant :

```
int execle(const char * app, const char * arg, ..., const char * envp []);
```

dans lequel on fournit un tableau explicite pour l'environnement désiré, comme avec `execve()`.

Récapitulons les caractéristiques des six fonctions de la famille `exec()`.

- `execv()`
  - tableau `argv[]` pour les arguments
  - variable externe globale pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execve()`
  - tableau `argv[]` pour les arguments
  - tableau `envp[]` pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execvp()`
  - tableau `argv[]` pour les arguments
  - variable externe globale pour l'environnement
  - application recherchée suivant le contenu de la variable `PATH`
- `execl()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - variable externe globale pour l'environnement
  - nom d'application avec chemin d'accès complet

- `execle()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - tableau `envp[]` pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execlp()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - variable externe globale pour l'environnement
  - application recherchée suivant le contenu de la variable `PATH`

Lorsqu'un processus exécute un appel `exec()` et que celui-ci réussit, le nouveau programme remplace totalement l'ancien. Les segments de données, de code, de pile sont réinitialisés. En conséquence, les variables allouées en mémoire sont automatiquement libérées. Les chaînes d'environnement et d'argument sont copiées ; on peut donc utiliser n'importe quel genre de variables (statiques ou allouées dynamiquement, locales ou globales) pour transmettre les arguments de l'appel `exec()`.

L'ancien programme transmet automatiquement au nouveau programme :

- Les PID et PPID, PGID et SID. Il n'y a donc pas de création de nouveau processus.
- Les identifiants UID et GID, sauf si le nouveau programme est Set-UID ou Set-GID. Dans ce cas, seuls les UID ou GID réels sont conservés, les identifiants effectifs étant mis à jour.
- Le masque des signaux bloqués, et les signaux en attente.
- La liste des signaux ignorés. Un signal ayant un gestionnaire installé reprend son comportement par défaut. Nous discuterons de ce point dans le chapitre 7.
- Les descripteurs de fichiers ouverts ainsi que leurs éventuels verrous, sauf si le fichier dispose de l'attribut *close-on-exec* ; dans ce cas, il est refermé.

En revanche :

- Les temps d'exécution associés au processus ne sont pas remis à zéro.
- Les privilèges du nouveau programme dérivent des précédents comme nous l'avons décrit dans le chapitre 2.

Causes d'échec de lancement d'un programme

Nous avons dit que lorsque l'appel `exec()` réussit, il ne revient pas. Lorsque le programme lancé se finit par `exit()`, `abort()` ou `return` depuis la fonction `main()`, le processus est terminé. Par conséquent, lorsque `exec()` revient dans le processus appelant, une erreur s'est produite. Il est important d'analyser alors le contenu de la variable globale `errno` afin d'expliquer le problème à l'utilisateur. Le détail en est fourni dans la page de manuel de l'appel `exec()` considéré. Voyons les types d'erreurs pouvant se produire :

- Le fichier n'existe pas, n'est pas exécutable, le processus appelant n'a pas les autorisations nécessaires, ou l'interpréteur requis n'est pas accessible : `EACCES`, `EPERM`, `ENOEXEC`, `ENOENT`, `ENOTDIR`, `EINVAL`, `EISDIR`, `ELIBBAD`, `ENAMETOOLONG`, `ELOOP`. Le programme doit alors détailler l'erreur avant de proposer à l'utilisateur une nouvelle tentative d'exécution.
- Le fichier est trop gros, la mémoire manque, ou un problème d'ouverture de fichier se pose : `E2BIG`, `ENOMEM`, `EIO`, `ENFILE`, `EMFILE`. On peut considérer cela comme une erreur critique, où le programme doit s'arrêter, après avoir expliqué le problème à l'utilisateur.

- Un pointeur est invalide : `EFAULT`. Il s'agit d'un bogue de programmation.
- Le fichier est déjà ouvert en écriture : `ETXTBSY`.

Pour pouvoir détailler un peu cette dernière erreur, nous devons nous intéresser à la méthode employée par Linux pour gérer la mémoire virtuelle.

L'espace mémoire dont dispose un processus est découpé en pages. Ces pages mesurent 4 Ko sur les systèmes à base de 80x86, mais varient suivant les architectures des machines. Leur dimension est définie dans `<asm/param.h>`. Les processus ont l'impression d'avoir un espace d'adressage linéaire et continu, mais en réalité le noyau peut déplacer les pages à son gré dans la mémoire physique du système. Une collaboration entre le noyau et le processeur permet d'assurer automatiquement la traduction d'adresse nécessaire lors d'un accès mémoire.

Une page peut également ne pas se trouver en mémoire, mais résider sur le disque. Lorsque le processus tente d'y accéder, le processeur déclenche une *faute de page* et le noyau charge à ce moment la page désirée. Cela permet d'économiser la mémoire physique vraiment disponible.

Parallèlement, lorsque le noyau a besoin de trouver de la place en mémoire, il élimine une ou plusieurs pages qui ont peu de chances d'être utilisées dans un avenir proche. Si la page à supprimer a été modifiée par le processus, il est nécessaire de la sauvegarder sur le disque. Le noyau utilise alors la zone de *swap*. Si, au contraire, la page n'a pas été changée depuis son premier chargement sur le disque, on peut l'éliminer sans problème, le noyau sait où la retrouver.

Nous découvrons là une grande force de cette gestion mémoire : le code exécutable d'un programme, n'étant jamais modifié par le processus, n'a pas besoin d'être chargé entièrement en permanence. Le noyau peut relire sur le disque les pages de code nécessaires au fur et à mesure de l'exécution du programme. Il faut donc s'assurer qu'aucun autre processus ne risque de modifier le fichier exécutable. Pour cela, le noyau le verrouille, et toute tentative d'ouverture en écriture d'un fichier en cours d'exécution se soldera par un échec.

Un scénario classique pour un développeur met en avant ce phénomène : on utilise simultanément plusieurs consoles virtuelles ou plusieurs Xterm, en répartissant l'éditeur de texte sur une fenêtre, le compilateur sur une seconde, et le lancement du programme en cours de travail sur la troisième. Cela permet de relancer la compilation en utilisant simplement la touche de rappel de l'historique du shell, et de redémarrer le programme développé de la même manière dans une autre fenêtre. On apporte une modification au programme, et on oublie de le quitter avant de relancer la compilation. Le compilateur échouera alors en indiquant qu'il ne peut pas écrire sur un fichier exécutable en cours d'utilisation.

De la même façon, il n'est pas possible de lancer un programme dont le fichier est déjà ouvert en écriture par un autre processus. Dans ce cas, l'erreur `ETXTBSY` se produit. Il est bon dans ce cas de prévenir l'utilisateur. Le message peut même lui indiquer de se reporter à la commande `fuser` pour savoir quel processus a ouvert le fichier en question.

Nous allons mettre en lumière ce principe dans le programme suivant, `exemple_execv`, qui tente – vainement – d'ouvrir en écriture son propre fichier exécutable. Il ouvre ensuite en mode d'ajout en fin de fichier `exemple_execvp` que nous avons créé plus haut. Le fait d'ouvrir ce fichier en mode d'ajout évite de détruire les informations qu'il contient. Il tente alors de l'exécuter.

exemple_execv.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (int argc, char * argv[])
{
    int fd;

    char * nv_argv[] = { "./exemple_execvp", (char *) NULL };

    fprintf(stdout, "Essai d'ouverture de %s ... ", argv[0]);

    if ((fd = open(argv[0], O_WRONLY | O_APPEND)) < 0) {
        if (errno != ETXTBSY) {
            fprintf(stdout, "impossible, errno %d\n", errno);
            exit(1);
        }
        fprintf(stdout, "échec ETXTBSY, fichier déjà utilisé \n");
    }

    fprintf(stdout, "Ouverture de exemple_execvp en écriture ... ");

    if ((fd = open("exemple_execvp", O_WRONLY | O_APPEND)) < 0) {
        fprintf(stdout, "impossible, errno %d\n", errno);
        exit(1);
    }

    fprintf(stdout, "ok \n Tentative d'exécuter exemple_execvp ... ");
    execv("./exemple_execvp", nv_argv);

    if (errno == ETXTBSY)
        fprintf(stdout, "échec ETXTBSY fichier déjà utilisé \n");
    else
        fprintf(stdout, "errno = %d\n", errno);
    return 1;
}
```

Comme on pouvait s'y attendre, le programme n'arrive pas à ouvrir en écriture un fichier en cours d'exécution ni à lancer un programme dont le fichier est ouvert.

```
$ ls
exemple_execlp    exemple_execv    exemple_execve    exemple_execvp
exemple_execlp.c  exemple_execv.c  exemple_execve.c  exemple_execvp.c
$ ./exemple_execv
Essai d'ouverture ./exemple_execv ... échec ETXTBSY, fichier déjà utilisé
Ouverture de exemple_execvp en écriture ... ok
Tentative d'exécuter exemple_execvp ... échec ETXTBSY fichier déjà utilisé
$
```

Fonctions simplifiées pour exécuter un sous-programme

Il y a de nombreux cas où on désire lancer une commande externe au programme, sans pour autant remplacer le processus en cours. On peut par exemple avoir une application principale qui lance des sous-programmes indépendants, ou désirer faire appel à une commande système. Dans ce dernier cas, on peut classiquement invoquer la commande `mail` pour transmettre un message à l'utilisateur, à l'administrateur, ou envoyer un rapport de bogue au concepteur du programme.

Pour cela, nous disposons de la fonction `system()` et de la paire `popen()` / `pclose()`, qui sont implémentées dans la bibliothèque C en invoquant `fork()` et `exec()` selon les besoins.

La fonction **`system()`** est déclarée ainsi dans `<stdlib.h>` :

```
int system (const char * commande);
```

Cette fonction invoque le shell en lui transmettant la commande fournie, puis revient après la fin de l'exécution. Pour ce faire, il faut exécuter un `fork()`, puis le processus lance la commande en appelant le shell « `/bin/sh -c commande` », tandis que le processus père attend la fin de son fils. Si l'invocation du shell échoue, `system()` renvoie 127. Si une autre erreur se produit, elle renvoie –1, sinon elle renvoie la valeur de retour de la commande exécutée. Une manière simplifiée d'implémenter `system()` pourrait être la suivante :

```
    int
notre_system (const char * commande)
{
    char *   argv[4];
    int      retour;
    pid_t    pid;

    if ((pid = fork()) < 0)
        /* erreur dans fork */
        return -1;

    if (pid == 0) {
        /* processus fils */
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = commande;
        argv[3] = (char *) NULL;
        execv("/bin/sh", argv);
        /* execv a échoué */
        exit(127);
    }

    /* processus père */
    /* attente de la fin du processus fils */
    while (waitpid(pid, & retour, 0) < 0)
        if (errno != EINTR)
            return -1;
    return retour;
}
```

Attention
La fonction `system()` représente une énorme faille de sécurité dans toute application installée Set-UID. Voyons le programme simple suivant :

exemple_system.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    system("ls");
    return 0;
}
```

Le programme ne fait que demander au shell d'exécuter `"ls"`. Pourtant, si on l'installe Set-UID *root*, il s'agit d'une faille de sécurité. En effet, lorsque le shell recherche la commande `"ls"`, il parcourt les répertoires mentionnés dans la variable d'environnement `PATH`. Celle-ci est héritée du processus père et peut donc être configurée par l'utilisateur pour inclure en premier le répertoire `"."`. Le shell exécutera alors de préférence la commande `"ls"` qui se trouve dans le répertoire en cours. Il suffit que l'utilisateur crée un shell script exécutable, et le tour est joué. Voyons un exemple, en créant le shell script suivant :

ls :

```
#! /bin/sh
echo faux ls
echo qui lance un shell
sh
```

Examinons l'exécution suivante :

```
$ ./exemple_system
exemple_execlp    exemple_execv.c   exemple_execvp    exemple_system.c
exemple_execlp.c  exemple_execve    exemple_execvp.c  ls
exemple_execv     exemple_execve.c  exemple_system
$ export PATH=.:$PATH
$ ./exemple_system
faux ls
qui lance un shell
$ exit
$
```

Tout d'abord, le programme s'exécute normalement et invoque « `sh -c ls` », qui trouve `ls` dans le répertoire `/bin` comme d'habitude. Ensuite, nous modifions notre `PATH` pour y placer en première position le répertoire en cours. À ce moment, le shell exécutera notre `"ls"` piégé qui lance un shell. Jusque-là, rien d'inquiétant. Mais imaginons maintenant que le programme soit Set-UID *root*. C'est ce que nous configurons avant de revenir en utilisateur normal.

```
$ su
Password:
# chown root.root exemple_system
```

```
# chmod +s exemple_system
# exit
$
```

À ce moment, l'exécution du programme lance le "ls" piégé avec l'identité de *root* !

```
$ ./exemple_system
faux ls
qui lance un shell
#
```

Comme nous avons inclus dans notre script une invocation de shell, nous nous retrouvons avec un shell connecté sous *root* ! Il ne faut pas s'imaginer que le fait de forcer la variable d'environnement PATH dans le programme aurait résolu le problème. D'autres failles de sécurité classiques existent, notamment en faussant la variable d'environnement IFS qui permet au shell de séparer ses arguments (normalement des espaces, des tabulations, etc.).

Il ne faut donc jamais employer la fonction system() dans un programme Set-UID (ou Set-GID). On peut utiliser à la place les fonctions exec(), qui ne parcourent pas les répertoires du PATH. Le vrai danger avec system() est qu'il appelle le shell au lieu de lancer la commande directement.

La véritable version de system(), présente dans la GlibC, est légèrement plus complexe puisqu'elle gère les signaux SIGINT et SIGQUIT (en les ignorant) et SIGCHLD (en le bloquant).

En théorie, le fait de transmettre une commande NULL sert à vérifier la présence du shell /bin/sh. Normalement, system() doit renvoyer une valeur non nulle s'il est bien là. En pratique, sous Linux, la vérification n'a pas lieu, GlibC considère que /bin/sh appartient au minimum vital d'un système Unix.

Après avoir bien compris que la fonction system() ne doit jamais être employée dans un programme Set-UID ou Set-GID, rien n'empêche de l'utiliser dans des applications simples ne nécessitant pas de privilèges. L'exemple que nous invoquions précédemment concernant l'appel de l'utilitaire mail est pourtant difficile à utiliser avec la fonction system(), car il faudrait d'abord créer un fichier contenant le message, puis lancer mail avec une redirection d'entrée.

Pour cela, il est plus pratique d'utiliser la fonction **popen()**, qui permet de lancer un programme à la manière de system(), mais en fournissant un des flux d'entrée ou de sortie standard pour dialoguer avec le programme appelant.

Le prototype de cette fonction, dans <stdio.h>, est le suivant :

```
FILE * popen (const char * commande, const char * mode);
```

La commande est exécutée comme avec system() en invoquant fork() et exec(), mais, de plus, le flux d'entrée ou de sortie standard de la commande est renvoyé au processus appelant. La chaîne de caractères mode doit contenir soit r (*read*), si on souhaite lire les données de la sortie standard de la commande dans le flux renvoyé, w (*write*) si on préfère écrire sur son entrée standard. Le flux renvoyé par la fonction popen() est tout à fait compatible avec les fonctions d'entrée-sortie classiques telles fprintf(), fscanf(), fread() ou fwrite(). Par contre, le flux doit toujours être refermé en utilisant la fonction pclose() à la place de fclose().

Lorsqu'on appelle **pclose()**, cette fonction attend que le processus exécutant la commande se termine, puis renvoie son code de retour.

Voici un exemple simple dans lequel nous avons exécuté la commande "mail", suivie de notre nom d'utilisateur obtenu avec getlogin(). La commande est exécutée en redirigeant son flux d'entrée standard. Nous pouvons donc écrire notre message tranquillement par une série de fprintf().

exemple_popen_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    FILE * message;
    char * commande;

    if ((commande = malloc(strlen(getlogin()) + 6)) == NULL) {
        fprintf(stderr, "Erreur malloc %d\n", errno);
        exit(1);
    }

    strcpy(commande, "mail ");
    strcat(commande, getlogin());

    if ((message = popen(commande, "w")) == NULL) {
        fprintf(stderr, " Erreur popen %d \n", errno);
        exit(1);
    }

    fprintf(message, "Ceci est un message \n");
    fprintf(message, "émis par moi-meme\n");

    pclose(message);

    return 0;
}
```

Lorsqu'il est lancé, ce programme émet bien le mail prévu. On notera que popen() effectue, comme system(), un execl() de /bin/sh -c commande. Cette fonction est donc recherchée dans les répertoires mentionnés dans le PATH.

Une autre application classique de popen(), utilisant l'entrée standard de la commande exécutée, est d'invoquer le programme indiqué dans la variable d'environnement PAGER, ou si elle n'existe pas, less ou more. Ces utilitaires affichent les données qu'on leur envoie page par page, en s'occupant de gérer la taille de l'écran (less permet même de revenir en arrière). C'est un moyen simple et élégant de fournir beaucoup de texte à l'utilisateur en lui laissant la possibilité de le consulter à sa guise.

Notre second exemple va lire la sortie standard de la commande exécutée. C'est une méthode généralement utilisée pour récupérer les résultats d'une application complémentaire ou pour

invoquer une commande système qui fournit des données difficiles à obtenir directement (`who`, `ps`, `last`, `netstat`…).

Nous allons ici invoquer la commande `ifconfig` en lui demandant l'état de l'interface réseau `eth0`. Si celle-ci est activée, `ifconfig` renvoie une sortie du genre :

```
eth0 Lien encap:Ethernet  HWaddr 00:50:04:8C:7A:ED
     inet adr:172.16.15.16  Bcast:172.16.255.255  Masque:255.255.0.0
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     Paquets Reçus:0 erreurs:0 jetés:0 débordements:7395 trames:0
     Paquets transmis:29667 erreurs:0 jetés:0 débordements:0 carrier:22185
     collisions:7395 lg file transmission:100
     Interruption:3 Adresse de base:0x200
```

Si `eth0` est désactivée, on obtient :

```
eth0 Lien encap:Ethernet  HWaddr 00:50:04:8C:7A:ED
     inet adr:172.16.15.16  Bcast:172.16.255.255  Masque:255.255.0.0
     BROADCAST MULTICAST  MTU:1500  Metric:1
     Paquets Reçus:0 erreurs:0 jetés:0 débordements:11730 trames:0
     Paquets transmis:47058 erreurs:0 jetés:0 débordements:0 carrier:35190
     collisions:11730 lg file transmission:100
     Interruption:3 Adresse de base:0x200
```

(Remarquez la différence dans la troisième ligne, `UP` dans un cas, et pas dans l'autre.) Si l'interface n'existe pas, `ifconfig` ne renvoie rien sur sa sortie standard, mais écrit un message :

```
eth0: erreur lors de la recherche d'infos sur l'interface: Périphérique non trouvé
```

sur sa sortie d'erreur.

Notre programme va donc lancer la commande et rechercher si une ligne de la sortie standard commence par `UP`. Si c'est le cas, il indique que l'interface est active. S'il ne trouve pas cette chaîne de caractères ou si la commande ne renvoie aucune donnée sur sa sortie standard, il considère l'interface comme étant inactive.

exemple_popen_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    FILE * sortie;
    char ligne [128];
    char etat  [128];

    if ((sortie = popen("/sbin/ifconfig eth0", "r")) == NULL) {
        fprintf(stderr, " Erreur popen %d \n", errno);
        exit(1);
    }
```

```
    while (fgets(ligne, 127, sortie) != NULL) {
        if (sscanf(ligne, "%s", etat) == 1)
            if (strcmp(etat, "UP") == 0) {
                fprintf(stderr, "interface eth0 en marche \n");
                pclose(sortie);
                return 0;
            }
    }
    fprintf(stdout, "interface eth0 inactive \n");
    pclose(sortie);
    return 0;
}
```

Cet exemple (un peu artificiel, convenons-en) montre quand même l'utilité d'invoquer une commande système et d'en récupérer aisément les informations. Encore une fois, insistons sur le manque de sécurité qu'offre `popen()` pour un programme susceptible d'être installé Set-UID ou Set-GID.

Un dernier exemple concernant `popen()` nous permet d'invoquer un script associé à l'application principale. Ce script est écrit en langage Tcl/Tk, et offre une boîte de saisie configurable. Il utilise les chaînes de caractères transmises en argument en ligne de commande :

- Le premier argument correspond au nom de la boîte de saisie (le titre de la fenêtre).
- Le second argument est le libellé affiché pour questionner l'utilisateur.
- Le troisième argument (éventuel) est la valeur par défaut pour la zone de saisie.

En invoquant ainsi ce script dans un Xterm :

```
$ ./exemple_popen_3.tk Approximation "Entrez le degré du polynôme pour
 l'approximation des trajectoires" 3
```

La fenêtre suivante apparaît :

Figure 4.1
Fenêtre de saisie en Tcl/Tk

Lorsqu'on appuie sur le bouton Ok, la valeur saisie est affichée sur la sortie standard.

Le script Tcl/Tk est volontairement simplifié ; il ne traite aucun cas d'erreur.

exemple_popen_3.tk :

```
#! /usr/bin/wish

## Le titre de la fenêtre est le premier argument reçu
## sur la ligne de commande.
wm title . [lindex $argv 0]
```

```
## Le haut de la boîte de dialogue contient un libellé
## fourni en second argument de la ligne de commande, et
## une zone de saisie dont le contenu par défaut est
## éventuellement fourni en troisième argument.
frame .haut -relief flat -borderwidth 2
label .libelle -text [lindex $argv 1]
entry .saisie -relief sunken -borderwidth 2
.saisie insert 0 [lindex $argv 2]
pack .libelle .saisie -in .haut -expand true -fill x

## Le bas contient deux boutons, Ok et Annuler, chacun avec
## sa procédure associée.
frame .bts -relief sunken -borderwidth 2
button .ok -text "Ok" -command bouton_ok
button .annuler -text "Annuler" -command bouton_annuler
pack .ok .annuler -side left -expand true -pady 3 -in .bts
pack .haut .bts
update

proc bouton_ok {} {
    ## La procédure associée à OK transmet la chaîne lue
    ## sur la sortie standard.
    puts [.saisie get]
    exit 0
}

proc bouton_annuler {} {
    ## Si on annule, on n'écrit rien sur la sortie standard.
    ## On quitte simplement.
    exit 0
}
```

Notre programme C va invoquer le script et traiter quelques cas d'échec, notamment en testant le code de retour de `pclose()`. Si une erreur se produit, on effectue la saisie à partir de l'entrée standard du processus. Ceci permet d'utiliser le même programme dans un environnement X-Window avec une boîte de dialogue ou sur une console texte avec une saisie classique.

La ligne de commande que `popen()` invoque est la suivante :

```
./exemple_popen_3.tk Saisie "Entrez votre nom" nom_login 2> /dev/null
```

dans laquelle *nom_login* est obtenu par la commande `getlogin()`.

On redirige la sortie d'erreur standard vers `/dev/null` afin d'éviter les éventuels messages d'erreur de Tk si on se trouve sur une console texte (on suppose que le shell `/bin/sh` utilisé par `popen()` est du type Bourne, ce qui est normalement le cas sous Linux). La chaîne "Entrez votre nom" est encadrée par des guillemets pour qu'elle ne constitue qu'un seul argument de la ligne de commande.

Voici le programme C qui invoque le script décrit précédemment :

exemple_popen_3.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>

    int
main (void)
{
    FILE * saisie;
    char * login ;
    char nom [128];
    char commande [128];

    if ((login = getlogin()) == NULL)
        strcpy(nom, "\"\"") ;
    else
        strcpy(nom, login) ;
    sprintf(commande, "./exemple_popen_3.tk "
                      "Saisie "
                      "\"Entrez votre nom\" "
                      "%s 2>/dev/null", nom);

    if ((saisie = popen(commande , "r")) == NULL) {
        /* Le script est, par exemple, introuvable */
        /* On va essayer de lire sur stdin.        */
        fprintf(stdout, "Entrez votre nom : ");
        if (fscanf(stdin, "%s", nom) != 1) {
            /* La lecture sur stdin échoue...         */
            /* On utilise une valeur par défaut.      */
            strcpy(nom, getlogin());
        }
        fprintf(stdout, "Nom saisi : %s\n", nom);
        return 0;
    }

    if (fscanf(saisie, "%s", nom) != 1) {
        if (pclose(saisie) != 0) {
            /* Le script a échoué pour une raison quelconque. */
            /* On recommence la saisie sur stdin.             */
            fprintf(stdout, "Entrez votre nom : ");
            if (fscanf(stdin, "%s", nom) != 1) {
                /* La lecture sur stdin échoue...         */
                /* On utilise une valeur par défaut.      */
                strcpy(nom, getlogin());
            }
        } else {
            /* L'utilisateur a cliqué sur Annuler. Il faut    */
            /* abandonner l'opération en cours.               */
```

```
            fprintf(stdout, "Pas de nom fourni - abandon\n");
            return 1;
        }
    } else {
        pclose(saisie);
    }
    fprintf(stdout, "Nom saisi : %s\n", nom);
    return 0;
}
```

Conclusion

Ce chapitre nous a permis de découvrir plusieurs méthodes pour lancer une application. Les mécanismes à base de `exec()` permettent de remplacer totalement le programme en cours par un autre qui est exécutable, tandis que les fonctions `system()` et `popen()-pclose()` servent plutôt à utiliser une autre application comme sous-programme de la première.