Programmation système en C sous Linux

Christophe Blaess

Éditions Eyrolles

ISBN : 2-212-09136-2

2000

2

La notion de processus

Nous allons commencer notre étude de la programmation en C sous Linux par plusieurs chapitres analysant les divers aspects de l'exécution des applications. Ce chapitre introduira la notion de processus, ainsi que les différents identifiants qui y sont associés, leurs significations et leurs utilisations dans le système.

Dans les chapitres suivants, nous étudierons les interactions qu'un processus peut établir avec son environnement, c'est-à-dire sa propre vision du système, configurée par l'utilisateur, puis l'exécution et la fin des processus, en analysant toutes les méthodes permettant de démarrer un autre programme, de suivre ou de contrôler l'utilisation des ressources, de détecter et de gérer les erreurs.

Présentation des processus

Sous Unix, toute tâche qui est en cours d'exécution est représentée par un **processus**. Un processus est une entité comportant à la fois des données et du code. On peut considérer un processus comme une unité élémentaire en ce qui concerne l'activité sur le système.

On peut imaginer un processus comme un programme en cours d'exécution. Cette représentation est très imparfaite car une application peut non seulement utiliser plusieurs processus concurrents, mais un unique processus peut également lancer l'exécution d'un nouveau programme, en remplaçant entièrement le code et les données du programme précédent.

À un instant donné, un processus peut, comme nous le verrons plus loin, se trouver dans divers états. Le noyau du système d'exploitation est chargé de réguler l'exécution des processus afin de garantir à l'utilisateur un comportement multitâche performant. Le noyau fournit un mécanisme de régulation des tâches qu'on nomme « ordonnancement » (en anglais *scheduling*). Cela assure la répartition équitable de l'accès au microprocesseur par les divers processus concurrents.

Sur une machine uni-processeur, il n'y a qu'un seul processus qui s'exécute effectivement à un instant donné. Le noyau assure une commutation régulière entre tous les processus

présents sur le système pour garantir un fonctionnement multitâche. Sur une machine multiprocesseur, le principe est le même, à la différence que plusieurs processus – mais rarement tous – peuvent s'exécuter réellement en parallèle.

On peut examiner la liste des processus présents sur le système à l'aide de la commande `ps`, et plus particulièrement avec ses options `ax`, qui nous permettent de voir les processus endormis, et ceux qui appartiennent aux autres utilisateurs. On voit alors, même sur un système apparemment au repos, une bonne trentaine de processus plus ou moins actifs :

```
$ ps ax
PID TTY STAT  TIME COMMAND
    1  ?  S    0:03 init
    2  ?  SW   0:03 (kflushd)
    3  ?  SW<  0:00 (kswapd)
    4  ?  SW   0:00 (nfsiod)
    5  ?  SW   0:00 (nfsiod)
    6  ?  SW   0:00 (nfsiod)
    7  ?  SW   0:00 (nfsiod)
   28  ?  S    0:00 /sbin/kerneld
  194  ?  S    1:26 syslogd
  203  ?  S    0:00 klogd
  225  ?  S    0:00 crond
  236  ?  SW   0:00 (inetd)
  247  ?  SW   0:00 (lpd)
  266  ?  S    0:00 (sendmail)
  278  ?  S    0:00 gpm -t PS/2
  291   1 SW   0:00 (mingetty)
    [...]
  626  ?  SW   0:00 (axnet)
25896  ?  SW   0:00 (.xsession)
25913  ?  S    0:15 xfwm
25915  ?  S    0:04 /usr/X11R6/bin/xfce 8 4 /var/XFCE/system.xfwmrc 0 8
25918  ?  S    0:03 /usr/X11R6/bin/xfsound 10 4 /var/XFCE/system.xfwmrc
25919  ?  S    0:00 /usr/X11R6/bin/xfpager 12 4 /var/XFCE/system.xfwmrc
29434  ?  S    1:56 /usr/local/applix/axdata/axmain -helper 29430 6 10 -
29436  ?  SW   0:00 (applix)
29802  p0 S    0:00 -bash
29970  ?  S    0:00 xplaycd
29978  ?  S    0:00 xmixer
30550  p1 S    0:00 -bash
31144  p1 R    0:00 ps ax
$
```

La commande `ps` affiche plusieurs colonnes dont la signification ne nous importe pas pour le moment. Retenons simplement que nous voyons en dernière colonne l'intitulé complet de la commande qui a démarré le processus, et en première colonne un numéro d'identification qu'on nomme PID.

Identification par le PID

Le premier processus du système, `init`, est créé directement par le noyau au démarrage. La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel-système **`fork()`**, qui va dupliquer le processus appelant. Au retour de cet appel-système, deux processus iden-

tiques continueront d'exécuter le code à la suite de `fork()`. La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original, qu'on nomme traditionnellement le processus père, et la nouvelle copie, le processus fils.

L'appel-système `fork()` est déclaré dans `<unistd.h>`, ainsi :

```
pid_t fork(void);
```

Les deux processus pouvant être distingués par leur numéro d'identification **PID** (*Process IDentifier*), il est possible d'exécuter deux codes différents au retour de l'appel-système `fork()`. Par exemple, le processus fils peut demander à être remplacé par le code d'un autre programme exécutable se trouvant sur le disque. C'est exactement ce que fait un shell habituellement.

Pour connaître son propre identifiant PID, on utilise l'appel-système **`getpid()`**, qui ne prend pas d'argument et renvoie une valeur de type `pid_t`. Il s'agit, bien entendu, du PID du processus appelant. Cet appel-système, déclaré dans `<unistd.h>`, est l'un des rares qui n'échouent jamais :

```
pid_t getpid (void);
```

Ce numéro de PID est celui que nous avons vu affiché en première colonne de la commande `ps`. La distinction entre processus père et fils peut se faire directement au retour de l'appel `fork()`. Celui-ci, en effet, renvoie une valeur de type `pid_t`, qui vaut zéro si on se trouve dans le processus fils, est négative en cas d'erreur, et correspond au PID du fils si on se trouve dans le processus père.

Voici en effet un point important : dans la plupart des applications courantes, la création d'un processus fils a pour but de faire dialoguer deux parties indépendantes du programme (à l'aide de signaux, de tubes, de mémoire partagée…). Le processus fils peut aisément accéder au PID de son père (noté PPID pour *Parent PID*) grâce à l'appel-système **`getppid()`**, déclaré dans `<unistd.h>` :

```
pid_t getppid (void);
```

Cette routine se comporte comme `getpid()`, mais renvoie le PID du père du processus appelant. Par contre, le processus père ne peut connaître le numéro du nouveau processus créé qu'au moment du retour du `fork()`[1].

On peut examiner la hiérarchie des processus en cours sur le système avec le champ PPID de la commande `ps axj` :

```
$ ps axj
 PPID   PID  PGID   SID TTY TPGID  STAT   UID   TIME COMMAND
    0     1     0     0  ?     -1  S       0   0:03 init
    1     2     1     1  ?     -1  SW      0   0:03 (kflushd)
    1     3     1     1  ?     -1  SW<     0   0:00 (kswapd)
    1     4     1     1  ?     -1  SW      0   0:00 (nfsiod)
    [...]
    1   296   296   296   6   296  SW      0   0:00 (mingetty)
  297   301   301   297  ?     -1  S       0  45:56 /usr/X11R6/bin/X
  297 25884 25884   297  ?     -1  S       0   0:00 (xdm)
```

1. En réalité, un processus pourrait établir la liste de ses fils en analysant le PPID de tous les processus en cours d'exécution, par exemple, à l'aide du pseudo-système de fichiers `/proc`, mais il est quand même beaucoup plus simple de mémoriser la valeur de retour de `fork()`.

```
25884 25896 25896   297  ?     -1  SW    500   0:00 (.xsession)
25896 25913 25896   297  ?     -1  S     500   0:15 xfwm
25913 25915 25896   297  ?     -1  S     500   0:04 /usr/X11R6/bin/xfce
25913 25918 25896   297  ?     -1  S     500   0:03 /usr/X11R6/bin/xfsou
25913 25919 25896   297  ?     -1  S     500   0:00 /usr/X11R6/bin/xfpag
25915 29801 25896   297  ?     -1  S       0   0:01 xterm -ls
```

On voit que `init` n'a pas de père (PPID = 0), mais qu'un grand nombre de processus héritent de lui. On peut observer également une filiation directe `xdm` (25884) – `.xsession` (25896) – `xfwm` (25913) – `xfce` (25915) – `xterm` (29801)…

Lorsqu'un processus est créé par `fork()`, il dispose d'une *copie* des données de son père, mais également de l'environnement de celui-ci et d'un certain nombre d'autres éléments (table des descripteurs de fichiers, etc.). On parle alors d'**héritage** du père.

Notons que, sous Linux, l'appel-système `fork()` est très économe car il utilise une méthode de « copie sur écriture ». Cela signifie que toutes les données qui doivent être dupliquées pour chaque processus (descripteurs de fichier, mémoire allouée…) ne seront pas immédiatement recopiées. Tant qu'aucun des deux processus n'a modifié des informations dans ces pages mémoire, il n'y en a qu'un seul exemplaire sur le système. Par contre, dès que l'un des processus réalise une écriture dans la zone concernée, le noyau assure la véritable duplication des données. Une création de processus par `fork()` n'a donc qu'un coût très faible en termes de ressources système.

En cas d'erreur, `fork()` renvoie la valeur -1, et la variable globale `errno` contient le code d'erreur, défini dans `<errno.h>`, ou plus exactement dans `<asm/errno.h>`, qui est inclus par le précédent fichier d'en-tête. Ce code d'erreur peut être soit `ENOMEM`, qui indique que le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus, soit `EAGAIN`, qui signale que le système n'a plus de place libre dans sa table des processus, mais qu'il y en aura probablement sous peu. Un processus est donc autorisé à réitérer sa demande de duplication lorsqu'il a obtenu un code d'erreur `EAGAIN`.

Voici à présent un exemple de création d'un processus fils par l'appel-système `fork()`.

exemple_fork.c :

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <errno.h>
    #include <sys/wait.h>

    int
main (void)
{
    pid_t   pid_fils;

    do {
        pid_fils = fork ();
    } while ((pid_fils == -1) && (errno == EAGAIN));

    if (pid_fils == -1) {
        fprintf (stderr, "fork () impossible, errno=%d\n", errno);
        return (1);
```

```
    }
    if (pid_fils == 0) {
        fprintf (stdout, "Fils : PID=%d, PPID=%d\n",
                getpid (), getppid ());
        return (0);
    } else {
        fprintf (stdout, "Père : PID=%d, PPID=%d\n",
                getpid (), getppid ());
        wait (NULL);
        return (0);
    }
}
```

Lors de son exécution, ce programme fournit les informations suivantes :

```
$ ./exemple_fork
Père : PID=31382, PPID=30550, PID fils=31383
Fils : PID=31383, PPID=31382
$
```

Le PPID du processus père correspond au shell.

Dans notre exemple, l'appel-système `fork()` boucle si le noyau n'a plus assez de place dans sa table interne pour créer un nouveau processus. Dans ce cas, le système est déjà probablement dans une situation assez critique, et il n'est pas utile de gâcher des ressources CPU en effectuant une boucle hystérique sur `fork()`. Il serait préférable d'introduire un délai d'attente dans notre code pour ne pas réitérer notre demande immédiatement, et attendre ainsi pendant quelques secondes que le système revienne dans un état plus calme. Nous verrons des moyens d'endormir le processus dans le chapitre 9.

On remarquera que nous avons introduit un appel-système `wait(NULL)` à la fin du code du père. Nous en reparlerons ultérieurement, mais on peut d'ores et déjà noter que cela permet d'attendre la fin de l'exécution du fils. Si nous n'avions pas employé cet appel-système, le processus père aurait pu se terminer avant son fils, redonnant la main au shell, qui aurait alors affiché son symbole d'invite (`$`) avant que le fils n'ait imprimé ses informations. Voici ce qu'on aurait pu observer :

```
$ ./exemple_fork
Père : PID=31397, PPID=30550, PID fils=31398
$ Fils : PID=31398, PPID=1
```

Identification de l'utilisateur correspondant au processus

À l'opposé des systèmes mono-utilisateurs (Dos, Windows 95/98…), un système Unix est particulièrement orienté vers l'identification de ses utilisateurs. Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées. Pour cela, chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus et qui est définie par une valeur numérique : l'**UID** (*User IDentifier*). Dans certaines situations que nous examinerons plus bas, il est nécessaire pour le processus de changer d'identité.

Il existe trois identifiants d'utilisateur par processus : l'UID **réel**, l'UID **effectif**, et l'UID **sauvé**. L'UID réel est celui de l'utilisateur ayant lancé le programme. L'UID effectif est celui

qui correspond aux privilèges accordés au processus. L'UID sauvé est une copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus.

L'essentiel des ressources sous Unix (données, périphériques…) s'exprime sous forme de nœuds du système de fichiers. Lors d'une tentative d'accès à un fichier, le noyau effectue des vérifications d'autorisation en prenant en compte l'UID effectif du processus appelant. Généralement, cet UID effectif est le même que l'UID réel (celui de la personne ayant invoqué le processus). C'est le cas de toutes les applications classiques ne nécessitant pas de privilège particulier, par exemple les commandes Unix classiques (`ls`, `cp`, `mv`…) qui s'exécutent sous l'identité de leur utilisateur, laissant au noyau le soin de vérifier les permissions d'accès.

Certaines applications peuvent toutefois avoir besoin – souvent ponctuellement – d'autorisations spéciales, tout en étant invoquées par n'importe quel utilisateur. L'exemple le plus évident est `su`, qui permet de changer d'identité, mais on peut en citer beaucoup d'autres, comme `mount`, qui peut autoriser sous Linux tout utilisateur à monter des systèmes de fichiers provenant d'un CD-Rom ou d'une disquette, par exemple. Il y a également les applications utilisant des couches basses des protocoles réseau comme `ping`. Dans ce cas, il faut que le processus garde son UID réel pour savoir qui agit, mais il dispose d'un UID effectif lui garantissant une liberté suffisante sur le système pour accéder aux ressources désirées.

Les appels-système **`getuid()`** et **`geteuid()`** permettent respectivement d'obtenir l'UID réel et l'UID effectif du processus appelant. Ils sont déclarés dans `<unistd.h>`, ainsi :

```
uid_t getuid  (void);
uid_t geteuid (void);
```

Le type `uid_t` correspondant au retour des fonctions `getuid()` et `geteuid()` est défini dans `<sys/types.h>`. Selon les systèmes, il s'agit d'un `unsigned int`, `unsigned short` ou `unsigned long`. Nous utilisons donc la conversion `%u` pour `fprintf()`, qui doit fonctionner dans la plupart des cas.

L'UID effectif est différent de l'UID réel lorsque le fichier exécutable dispose d'un attribut particulier permettant au processus de changer d'identité au démarrage du programme. Considérons par exemple le programme suivant.

exemple_getuid.c :

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

    int
main (void)
{
    fprintf (stdout, "  UID réel = %u, UID effectif = %u\n",
         getuid (), geteuid ());
    return (0);
}
```

Quand on compile ce programme, on obtient un fichier exécutable, qu'on lance ensuite :

```
$ ls -ln exemple_getuid*
  rwxrwxr-x   1 500      500      4446 Jun 10 10:56 exemple_getuid
  rw-rw-r--   1 500      500       208 Jun 10 10:56 exemple_getuid.c
$ ./exemple_getuid UID réel = 500, UID effectif = 500
$
```

Le comportement est pour l'instant parfaitement normal. Imaginons maintenant que *root* passe par là, s'attribue le fichier exécutable et lui ajoute le bit « **Set-UID** » à l'aide de la commande `chmod`. Lorsqu'un utilisateur va maintenant exécuter `exemple_getuid`, le système va lui fournir l'UID effectif du propriétaire du fichier, à savoir *root* (qui a toujours l'UID 0 par définition) :

```
$ su
Password:
# chown root.root exemple_getuid
# chmod +s exemple_getuid
# ls -ln exemple_getuid*
  rwsrwsr-x   1 0        0        4446 Jun 10 10:56 exemple_getuid
  rw-rw-r--   1 500      500       208 Jun 10 10:56 exemple_getuid.c
# ./exemple_getuid
UID réel = 0, UID effectif = 0
# exit
$ ./exemple_getuid
UID réel = 500, UID effectif = 0
$
```

Nous voyons l'attribut Set-UID indiqué par la lettre « `s` » dans les autorisations d'accès. L'UID réel est conservé à des fins d'identification éventuelle au sein du processus.

Notre programme ayant l'UID effectif de *root* en a tous les privilèges. Vous pouvez en avoir le cœur net en lui faisant, par exemple, créer un nouveau fichier dans le répertoire `/etc`. Si vous n'avez pas les privilèges *root* sur votre système, vous pouvez néanmoins effectuer les tests en accord avec un autre utilisateur qui copiera votre exécutable dans son répertoire personnel (pour en prendre possession) et lui ajoutera le bit Set-UID.

Il existe plusieurs appels-système permettant à un processus de modifier son UID. Il ne peut toutefois s'agir que de perdre des privilèges, éventuellement d'en retrouver des anciens, mais jamais d'en gagner. Imaginons un émulateur de terminal série (un peu comme `kermit` ou `minicom`). Il a besoin d'accéder à un périphérique système (le modem), même en étant lancé par n'importe quel utilisateur. Il dispose donc de son bit Set-UID activé, tout en appartenant à *root*. Cela lui permet d'ouvrir le fichier spécial correspondant au périphérique et de gérer la liaison.

Toutefois, il faut également sauvegarder sur disque des informations n'appartenant qu'à l'utilisateur ayant lancé l'application (sa configuration préférée pour l'interface, par exemple), voire enregistrer dans un fichier un historique complet de la session. Pour ce faire, le programme ne doit créer des fichiers que dans des endroits où l'utilisateur est autorisé à le faire. Plutôt que de vérifier toutes les autorisations d'accès, il est plus simple de perdre temporairement ses privilèges *root* pour reprendre l'identité de l'utilisateur original, le temps de faire l'enregistrement (les permissions étant alors vérifiées par le noyau), et de redevenir éventuellement *root* ensuite. Nous reviendrons à plusieurs reprises sur ce mécanisme.

Le troisième type d'UID d'un processus est l'UID sauvé. Il s'agit d'une copie de l'ancien UID effectif lorsque celui-ci est modifié par l'un des appels décrits ci-dessous. Cette copie est effectuée automatiquement par le noyau. Un processus peut toujours demander à changer son UID effectif ou son UID réel pour prendre la valeur de l'UID sauvé. Il est également possible de prendre en UID effectif la valeur de l'UID réel, et inversement.

Un processus avec le bit Set-UID positionné démarre donc avec un UID effectif différent de celui de l'utilisateur qui l'a invoqué. Quand il désire effectuer une opération non privilégiée, il peut demander à remplacer son UID effectif par l'UID réel. Une copie de l'UID effectif est conservée dans l'UID sauvé. Il pourra donc à tout moment demander à remplacer à nouveau son UID effectif par son UID sauvé.

Pour cela, il existe plusieurs appels-système permettant sous Linux de modifier son UID, et ayant tous une portabilité restreinte : **`setuid()`** est défini par Posix.1, **`seteuid()`**et **`setreuid()`**appartiennent à BSD, **`setresuid()`**est spécifique à Linux.

La philosophie des développeurs Linux est exprimée dans le fichier `/usr/src/linux/kernel/sys.c`, qui implémente tous ces appels-système. Un programme utilisant uniquement `seteuid()` ou `setreuid()` sera 100 % compatible avec BSD, un programme utilisant uniquement `setuid()` sera 100 % compatible avec Posix. Bien entendu, ils seront tous deux 100 % compatibles avec Linux !

Les trois premiers appels-système sont déclarés dans `<unistd.h>`, ainsi :

```
int setuid   (uid_t uid_effectif);
int seteuid  (uid_t uid_effectif);
int setreuid (uid_t uid_reel, uid_t uid_effectif);
```

Ils permettent de modifier un ou plusieurs UID du processus appelant, renvoyant 0 s'ils réussissent, ou -1 en cas d'échec.

Nous allons voir le comportement d'un programme Set-UID qui abandonne temporairement ses privilèges pour disposer des permissions de l'utilisateur l'ayant invoqué, puis qui reprend à nouveau ses autorisations originales. Notez bien que, dans cette première version, la récupération de l'ancienne identité *ne fonctionne pas* si le programme appartient à *root*. Ceci est clairement défini dans l'implémentation de `setuid()`. Les développeurs de Linux préviennent bien qu'en cas de mécontentement, il faut s'en prendre au comité Posix, qui est responsable de cette règle. Nous verrons immédiatement après une version utilisant `setreuid()`, qui fonctionne dans tous les cas de figure.

exemple_setuid.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t       uid_reel;
    uid_t       uid_eff;

    uid_reel = getuid ();
    uid_eff = geteuid ();
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setuid (%d) = %d\n", uid_reel, setuid (uid_reel));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setuid (%d) = %d\n", uid_eff, setuid (uid_eff));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
```

```
    return (0);
}
```

L'exécution du programme (copié par un autre utilisateur, et avec le bit Set-UID positionné) donne :

```
$ ls -ln exemple_setuid*
-rwsrwsr-x   1 501      501      4717 Jun 10 15:49 exemple_setuid
$ ./exemple_setuid
  UID réel = 500,  UID effectif = 501
 setuid (500) = 0
  UID réel = 500,  UID effectif = 500
 setuid (501) = 0
  UID réel = 500,  UID effectif = 501
$
```

Si on tente la même opération avec un programme Set-UID *root*, il ne pourra plus reprendre ses privilèges, car lorsque `setuid()` est invoqué par un utilisateur ayant un UID effectif nul (*root*), il écrase également l'UID sauvé pour empêcher le retour en arrière.

Voici maintenant une variante utilisant l'appel-système `setreuid()`. Comme on peut s'en douter, il permet de fixer les deux UID en une seule fois. Si l'un des deux UID vaut –1, il n'est pas changé. Cet appel-système n'est pas défini par Posix.1, mais appartient à l'univers BSD.

exemple_setreuid.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t       uid_reel;
    uid_t       uid_eff;

    uid_reel = getuid ();
    uid_eff = geteuid ();
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (-1, %d) = %d\n", uid_reel,
                        setreuid (-1, uid_reel));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (-1, %d) = %d\n", uid_eff,
                        setreuid (-1, uid_eff));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (%d, -1) = %d\n", uid_eff,
                        setreuid (uid_eff, -1));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());

    return (0);
}
```

En voici l'exécution, après passage en Set-UID *root* :

```
$ ls -ln exemple_setre*
  rwsrwsr-x   1 0        0        4809 Jun 10 16:23 exemple_setreuid
  rw-rw-r--   1 500      500       829 Jun 10 16:23 exemple_setreuid.c
$ ./exemple_setreuid
 UID-R = 500,  UID-E = 0
setreuid (-1, 500) = 0
 UID-R = 500,  UID-E = 500
setreuid (-1, 0) = 0
UID-R = 500,  UID-E = 0
setreuid (0, -1) = 0
UID-R = 0,  UID-E = 0
$
```

Cette fois-ci, le changement fonctionne parfaitement, même avec un UID effectif nul.

Enfin, il est possible – mais c'est une option spécifique à Linux – de modifier également l'UID sauvé, principalement pour empêcher le retour en arrière comme le fait `setuid()`, avec l'appel-système **`setresuid()`**. Celui-ci et l'appel-système complémentaire **`getresuid()`**ne sont définis que depuis les noyaux 2.2. Attention donc aux problèmes de portabilité. D'autant que l'appel-système est bien présent, mais les fichiers d'en-tête de la bibliothèque GlibC 2.1 ne proposent pas encore les prototypes correspondants :

```
int setresuid (uid_t uid_reel, uid_t uid_effectif, uid_t uid_sauve);
int getresuid (uid_t * uid_reel, uid_t * uid_effectif, uid_t * uid_sauve);
```

exemple_setresuid.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t       uid_reel = getuid  ();
    uid_t       uid_eff  = geteuid ();
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (-1, %d) = %d\n", uid_reel,
                         setreuid (-1, uid_reel));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (-1, %d) = %d\n", uid_eff,
                         setreuid (-1, uid_eff));
    fprintf (stdout, " UID-R = %u, UID-E = %u\n", getuid (), geteuid ());
    fprintf (stdout, " setreuid (%d, -1) = %d\n", uid_eff,
                         setreuid (uid_eff, -1));
    fprintf (stdout, " UID-R = %u,  UID-E = %u\n", getuid (), geteuid ());
    return (0);
}
```

L'exécution est intéressante si le programme est installé Set-UID *root* :

```
$ ls -ln exemple_setresuid
-rwsrwsr-x   1 0        0       12404 Nov 14 15:10 exemple_setresuid
```

```
$ ./exemple_setresuid
  UID-R = 500,  UID-E = 0, UID-S = 0
 setresuid (-1, 500, 0) = 0
  UID-R = 500,  UID-E = 500, UID-S = 0
 setresuid (-1, 0, -1) = 0
  UID-R = 500,  UID-E = 0, UID-S = 0
$
```

Identification du groupe d'utilisateurs du processus

Chaque utilisateur du système appartient à un ou plusieurs groupes. Ces derniers sont définis dans le fichier `/etc/groups`. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les UID, un processus dispose donc de plusieurs **GID** (*Group IDentifier*) réel, effectif, sauvé, ainsi que de GID supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.

ATTENTION Il ne faut pas confondre les *groupes d'utilisateurs* auxquels un processus appartient, et qui dépendent de la personne qui lance le processus et éventuellement des attributs Set-GID du fichier exécutable, avec les *groupes de processus*, qui permettent principalement d'envoyer des signaux à des ensembles de processus. Un processus appartient donc à deux types de groupes qui n'ont rien à voir les uns avec les autres.

Le GID réel correspond au groupe principal de l'utilisateur ayant lancé le programme (celui qui est mentionné dans `/etc/passwd`).

Le GID effectif peut être différent du GID réel si le fichier exécutable dispose de l'attribut **Set-GID** (`chmod g+s`). C'est le GID effectif qui est utilisé par le noyau pour vérifier les autorisations d'accès aux fichiers.

La lecture de ces GID se fait symétriquement à celle des UID avec les appels-système **`getgid()`**et **`getegid()`**. La modification (sous réserve d'avoir les autorisations nécessaires) peut se faire à l'aide des appels **`setgid()`, `setegid()`**et **`setregid()`**. Les fonctions `getgid()` et `setgid()` sont compatibles avec Posix.1, les autres avec BSD. Les prototypes de ces fonctions sont présents dans `<unistd.h>`, le type `gid_t` étant défini dans `<sys/types.h>` :

```
gid_t getgid (void);
gid_t getegid (void);
int setgid (gid_t egid);
int setegid (gid_t egid);
int setregid (gid_t rgid, gid_t egid);
```

Les deux premières fonctions renvoient le GID demandé, les deux dernières renvoient 0 si elle réussissent et -1 en cas d'échec.

L'ensemble complet des groupes auxquels appartient un utilisateur est indiqué dans `/etc/groups` (en fait, c'est une table inversée puisqu'on y trouve la liste des utilisateurs appartenant à chaque groupe). Un processus peut obtenir cette liste en utilisant l'appel-système **`getgroups()`** :

```
int getgroups (int taille, gid_t liste []);
```

Celui-ci prend deux arguments, une dimension et une table. Le premier argument indique la taille (en nombre d'entrées) de la table fournie en second argument. L'appel-système va remplir le tableau avec la liste des GID supplémentaires du processus. Si le tableau est trop petit, `getgroups()` échoue (renvoie –1 et remplit `errno`), sauf si la taille est nulle ; auquel cas,

il renvoie le nombre de groupes supplémentaires du processus. La manière correcte d'utiliser `getgroups()` est donc la suivante.

exemple_getgroups.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    int     taille;
    gid_t   * table_gid = NULL;
    int     i;

    if ((taille = getgroups (0, NULL)) < 0) {
        fprintf (stderr, "Erreur getgroups, errno = %d\n", errno);
        return (1);
    }

    if ((table_gid = calloc (taille, sizeof (gid_t))) == NULL) {
        fprintf (stderr, "Erreur  calloc, errno = %d\n", errno);
        return (1);
    }

    if (getgroups (taille, table_gid) < 0) {
        fprintf (stderr, "Erreur getgroups, errno = %d\n", errno);
        return (1);
    }
    for (i = 0; i < taille; i ++)
        fprintf (stdout, "%u ", table_gid [i]);
    fprintf (stdout, "\n");

    free (table_gid);
    return (0);
}
```

qui donne :

```
$ ./exemple_getgroups
500 100
$
```

Le nombre maximal de groupes auxquels un utilisateur peut appartenir est défini dans `<asm/param.h>` sous le nom **`NGROUPS`**. Cette constante symbolique vaut 32 par défaut sous Linux.

Il est possible de fixer sa liste de groupes supplémentaires. La fonction **`setgroups()`** n'est néanmoins utilisable que par *root* (ou un processus dont le fichier exécutable est Set-UID *root*)[1]. Contrairement à `getgroups()`, le prototype est inclus dans le fichier `<grp.h>` de la bibliothèque GlibC 2 :

```
int setgroups (size_t taille, const gid_t * table);
```

1. En réalité, depuis Linux 2.2, il suffit que le processus ait la capacité `CAP_SETGID` comme nous le verrons en fin de chapitre.

Il faut définir la constante symbolique _BSD_SOURCE pour avoir accès à cette fonction.
exemple_setgroups.c :

```
    #define __BSD_SOURCE

    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <errno.h>
    #include <grp.h>

    int
main (int argc, char * argv [])
{
    gid_t   * table_gid = NULL;
    int     i;
    int     taille;

    if (argc < 2) {
        fprintf (stderr, "Usage %s GID ...\n", argv [0]);
        return (1);
    }
    if ((table_gid = calloc (argc - 1, sizeof (gid_t))) == NULL) {
        fprintf (stderr, "Erreur calloc, errno = %d\n", errno);
        return (1);
    }
    for (i = 1; i < argc ; i ++)
        if (sscanf (argv [i], "%u", & (table_gid [i - 1])) != 1) {
            fprintf (stderr, "GID invalide : %s\n", argv [i]);
            return (1);
        }
    if (setgroups (i - 1, table_gid) < 0) {
        fprintf (stderr, "Erreur setgroups, errno = %d\n", errno);
        return (1);
    }
    free (table_gid);

    /* Passons maintenant à la vérification des groupes */
    fprintf (stdout, "Vérification : ");
    /*                ...
     * même code que la fonction main() de exemple_getgroups.c
     *                ...
     */
}
```

Ce programme ne fonctionne que s'il est Set-UID *root* :

```
$ ls -ln exemple_setgroups*
-rwxrwxr-x   1 500      500          5606 Jun 11 14:10 exemple_setgroups
-rw-rw-r--   1 500      500          1612 Jun 11 14:05 exemple_setgroups.c
$ ./exemple_setgroups 501 502 503
Erreur setgroups, errno = 1
```

```
$ su
Password:
# chown root.root exemple_setgroups
# chmod +s exemple_setgroups
# exit
$ ls -ln exemple_setgroups*
-rwsrwsr-x   1 0        0            5606 Jun 11 14:10 exemple_setgroups
-rw-rw-r--   1 500      500          1612 Jun 11 14:05 exemple_setgroups.c
$ ./exemple_setgroups 501 502 503
Vérification : 501 502 503
$
```

Pour un processus Set-UID *root*, le principal intérêt de la modification de la liste des groupes auxquels appartient un processus est de pouvoir ajouter un groupe spécial (donnant par exemple un droit de lecture et d'écriture sur un fichier spécial de périphérique) à sa liste, et de changer ensuite son UID effectif pour continuer à s'exécuter sous l'identité de l'utilisateur, tout en gardant le droit d'agir sur ledit périphérique.

Tout comme nous l'avons vu plus haut avec les UID, il existe sous Linux un GID sauvé pour chaque processus. Cela permet de modifier son GID effectif (en reprenant temporairement l'identité réelle), puis de retrouver le GID effectif original (qui était probablement fourni par le bit Set-GID). Pour accéder aux GID sauvés, deux appels-système, **`setresgid()`** et **`getresgid(),`** ont fait leur apparition dans le noyau Linux 2.2 :

```
int setresgid (gid_t uid_reel, uid_t uid_effectif, uid_t uid_sauve);
int getresgid (gid_t * uid_reel, * uid_t uid_effectif, * uid_t uid_sauve);
```

Le programme `exemple_setresgid.c` est une copie de `exemple_setresuid.c` dans lequel on a changé toutes les occurrences de `uid` en `gid`. En voici un exemple d'exécution après sa transformation en programme Set-GID *root* :

```
$ ls -ln ./exemple_setresgid
-rwxrwsr-x   1 0        0        12404 Nov 14 15:38 ./exemple_setresgid
$ ./exemple_setresgid
  GID-R = 500,  GID-E = 0, GID-S = 0
 setresgid (-1, 500, 0) = 0
  GID-R = 500,  GID-E = 500, GID-S = 0
 setresgid (-1, 0, -1) = 0
  GID-R = 500,  GID-E = 0, GID-S = 0
$
```

Identification du groupe de processus

Les processus sont organisés en groupes. Rappelons qu'il ne faut pas confondre les groupes de processus avec les groupes d'utilisateurs que nous venons de voir, auxquels appartiennent les processus. Les groupes de processus ont pour principale utilité de permettre l'envoi global de signaux à un ensemble de processus. Ceci est notamment utile aux interpréteurs de commandes, qui l'emploient pour implémenter le contrôle des jobs. Pour savoir à quel groupe appartient un processus donné, on utilise l'appel-système **`getpgid(),`** déclaré dans `<unistd.h>` :

```
pid_t   getpgid (pid_t pid);
```

Celui-ci prend en argument le PID du processus visé et renvoie son numéro de groupe, ou –1 si le processus mentionné n'existe pas. Avec la bibliothèque GlibC 2, `getpgid()` n'est défini dans `<unistd.h>` que si la constante symbolique `_GNU_SOURCE` est déclarée avant l'inclusion.

exemple_getpgid.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv [])
{
    int     i;
    pid_t   pid;
    pid_t   pgid;

    if (argc == 1) {
        fprintf (stdout, "%u : %u\n", getpid (), getpgid (0));
        return (0);
    }
    for (i = 1; i < argc; i ++)
        if (sscanf (argv [i], "%u", & pid) != 1) {
            fprintf (stderr, "PID invalide : %s\n", argv [i]);
        } else {
            pgid = getpgid (pid);
            if (pgid == -1)
                fprintf (stderr, "%u inexistant\n", pid);
            else
                fprintf (stderr, "%u : %u\n", pid, pgid);
        }
    return (0);
}
```

Ce programme permet de consulter les groupes de n'importe quels processus, « 0 » signifiant « processus appelant ».

```
$ ps
 PID TTY STAT TIME COMMAND
4519  p1 S  0:00 -bash
4565  p0 S  0:00 -bash
5017  p1 S  0:00 man getpgid
5018  p1 S  0:00 sh -c (cd /usr/man/fr_FR ; /usr/bin/gtbl /usr/man/fr_FR/m
5019  p1 S  0:00 sh -c (cd /usr/man/fr_FR ; /usr/bin/gtbl /usr/man/fr_FR/m
5022  p1 S  0:00 /usr/bin/less -is
5026  p0 R  0:00 ps
$ ./exemple_getpgid 4519 4565 5017 5018 5019 5022 5026 0
4519 : 4519
4565 : 4565
5017 : 5017
5018 : 5017
5019 : 5017
5022 : 5017
5026 inexistant
0 : 5027
$
```

Un groupe a été créé au lancement du processus 5017 (*man*), et il comprend tous les descendants (mise en forme et affichage de la page). Le processus dont le PID est identique au numéro de groupe est nommé **leader** du groupe. Un groupe n'a pas nécessairement de leader, celui-ci pouvant se terminer alors que ses descendants continuent de s'exécuter.

Il existe un appel-système **`getpgrp()`**, qui ne prend pas d'argument et renvoie le numéro de groupe du processus appelant, exactement comme `getpgid(0)`. Attention toutefois, la portabilité de cet appel-système n'est pas assurée, certaines versions d'Unix l'implémentant comme un synonyme exact de `getpgid()`.

exemple_getpgrp.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv [])
{
    fprintf (stdout, "%u : %u\n", getpid (), getpgrp ());
    return (0);
}
```

```
$ ./exemple_getpgrp
7344 : 7344
$
```

La plupart des applications n'ont pas à se préoccuper de leur groupe de processus, mais cela peut parfois être indispensable lorsqu'on désire envoyer un signal à tous les descendants d'un processus père. Les interpréteurs de commandes, ou les programmes qui lancent des applications diverses (gestionnaires de fenêtres X11, gestionnaires de fichiers…), doivent pouvoir tuer tous les descendants directs d'un processus fils. Cela peut aussi être nécessaire si l'application crée de nombreux processus fils (par exemple à chaque demande de connexion pour un démon serveur réseau) et désire pouvoir se terminer complètement en une seule fois.

Un processus peut modifier son propre identifiant de groupe ou celui de l'un de ses descendants grâce à l'appel-système **`setpgid()`** :

```
int setpgid (pid_t pid, pid_t pgid);
```

Le premier argument correspond au PID du processus à modifier. Si cet argument est nul, on considère qu'il s'agit du processus appelant. Le second argument indique le nouveau numéro de groupe pour le processus concerné. Si le second argument est égal au premier ou s'il est nul, le processus devient leader de son groupe.

L'appel-système échoue si le processus visé n'est ni le processus appelant ni l'un de ses descendants. Par ailleurs, un processus ne peut plus modifier le groupe de l'un de ses descendants si celui-ci a effectué un appel à l'une des fonctions de la famille `exec()`. Généralement, les interpréteurs de commandes utilisent la procédure suivante :

- Le shell exécute un `fork()`. Le processus père en garde le résultat dans une variable *pid_fils*.
- Le processus fils demande à devenir leader de son groupe en invoquant `setpgid(0,0)`.

- De manière redondante, le processus père réclame que son fils devienne leader de son groupe, cela pour éviter tout problème de concurrence d'exécution. Le père exécute donc `setpgid (pid_fils, pid_fils)`.
- Le père peut alors attendre, par exemple, la fin de l'exécution du fils avec `waitpid()`.
- Le fils appelle une fonction de la famille `exec()` pour lancer la commande désirée.

Le shell pourra alors contrôler l'ensemble des processus appartenant au groupe du fils en leur envoyant des signaux (`STOP`, `CONT`, `TERM`...).

Il existe un appel-système **`setpgrp()`**, qui sert directement à créer un groupe de processus et à en devenir leader. Il s'agit d'un synonyme de `setpgid (0, 0)`. Attention là encore à la portabilité de cet appel-système, car sous BSD il s'agit d'un synonyme de `setpgid()` utilisant donc deux arguments.

Identification de session

Il existe finalement un dernier regroupement de processus, les **sessions**, qui réunissent divers groupes de processus. Les sessions sont très liées à la notion de **terminal de contrôle**des processus. Il n'y a guère que les shells ou les gestionnaires de fenêtres pour les environnements graphiques qui ont besoin de gérer les sessions. Une exception toutefois : les applications qui s'exécutent sous forme de démon doivent accomplir quelques formalités concernant leur session. C'est donc principalement ce point de vue qui nous importera ici.

Généralement, une session est attachée à un terminal de contrôle, celui qui a servi à la connexion de l'utilisateur. Avec l'évolution des systèmes, les terminaux de contrôle sont souvent des pseudo-terminaux virtuels gérés par les systèmes graphiques de fenêtrage ou par les pilotes de connexion réseau, comme nous le verrons dans le chapitre 33. Au sein d'une session, un groupe de processus est en avant-plan ; il reçoit directement les données saisies sur le clavier du terminal, et peut afficher ses informations de sortie sur l'écran de celui-ci. Les autres groupes de processus de la session s'exécutent en arrière-plan. Leur interaction avec le terminal sera étudiée ultérieurement dans le chapitre sur les signaux.

Pour créer une nouvelle session, un processus ne doit **pas** être leader de son groupe. En effet, la création de la session passe par une étape de constitution d'un nouveau groupe de processus prenant l'identifiant du processus appelant. Il est indispensable que cet identifiant ne soit pas encore attribué à un groupe qui pourrait contenir éventuellement d'autres processus.

La création d'une session s'effectue par l'appel-système **`setsid()`**, déclaré dans `<unistd.h>` :

```
pid_t setsid (void);
```

Il renvoie le nouvel identifiant de session, de type `pid_t`. Lors de cet appel, un nouveau groupe est créé, il ne contient que le processus appelant (qui en est donc le leader). Puis, une nouvelle session est créée, ne contenant pour le moment que ce groupe. Cette session ne dispose pas de terminal de contrôle. Elle devra en récupérer un explicitement si elle le désire. Les descendants du processus leader se trouveront, bien entendu, dans cette nouvelle session.

Un point de détail reste à préciser. Pour être sûr que le processus initial n'est pas leader de son groupe, on utilise généralement l'astuce suivante :

- Un processus père exécute un `fork()`, suivi d'un `exit()`.

- Le processus fils se trouvant dans le même groupe que son père ne risque pas d'être leader, et peut donc tranquillement invoquer `setsid()`.

La fonction **`getsid()`** prend en argument un PID et renvoie l'identifiant de la session, c'est-à-dire le PID du processus leader :

```
pid_t getsid (pid_t pid);
```

Cet appel-système n'est déclaré dans `<unistd.h>` que si la constante `_GNU_SOURCE` est définie avant son inclusion. Cette fonction n'échoue que si le PID transmis ne correspond à aucun processus existant. Comme d'habitude, `getsid(0)` renvoie l'identifiant du processus appelant. Cette fonction n'est pas décrite dans Posix et ne sera pas portable sur les systèmes BSD.

exemple_getsid.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv [])
{
    int     i;
    pid_t   pid;
    pid_t   sid;

    if (argc == 1) {
        fprintf (stdout, "%u : %u\n", getpid (), getsid (0));
        return (0);
    }
    for (i = 1; i < argc; i ++)
        if (sscanf (argv [i], "%u", & pid) != 1) {
            fprintf (stderr, "PID invalide : %s\n", argv [i]);
        } else {
            sid = getsid (pid);
            if (sid == -1)
                fprintf (stderr, "%u inexistant\n", pid);
            else
                fprintf (stderr, "%u : %u\n", pid, sid);
        }
    return (0);
}
```

```
$ ps ax
...
  509 ?   SW   0:00 [kdm]
  521 ?   S    1:01 kwm
  538 ?   S    0:03 kbgndwm
  554 ?   S    2:36 /usr/bin/kswarm.kss -delay 3 -install -corners iiii -
  566 ?   S    0:43 kfm
  567 ?   S    0:01 krootwm
  568 ?   S    0:40 kpanel
  587 ?   SN   0:02 /usr/bin/kapm
```

```
  747 ?   SW   0:00 [axnet]
  748 ?   S   15:09 /usr/local/applix/axdata/axmain -helper
  750 ?   SW   0:00 [applix]
  758 ?   S    0:05 konsole -icon konsole.xpm -miniicon konsole.xpmi -cap
  759 ?   S    0:01 /bin/bash
  763 ?   SW   0:00 [gnome-name-serv]
$ ./exemple_getsid 0 567 748 521
0 : 759
567 : 501
748 : 501
521 : 501
$
```

Nous voyons que le processus en cours appartient à la session de son interpréteur de commandes (`/bin/bash`) et que les applications graphiques dépendent du serveur X11.

L'interaction entre un processus et un terminal s'effectue donc par l'intermédiaire de plusieurs indirections :

- Le processus appartient toujours à un groupe.
- Le groupe appartient à une session.
- La session peut – éventuellement – avoir un terminal de contrôle.
- Le terminal connaît le numéro du groupe de processus en avant-plan.

C'est en général le leader de session (le shell) qui assure le basculement en avant-plan ou en arrière-plan des groupes de processus de sa session, en utilisant les fonctions de dialogue avec le terminal, `tcgetpgrp()` et `tcsetpgrp()`. Ces fonctions seront analysées ultérieurement dans le chapitre 33.

Capacités d'un processus

Depuis Linux 2.2, la toute-puissance d'un processus exécuté sous l'UID effectif *root* peut être limitée. Une application dispose à présent d'un jeu de capacités permettant de définir ce que le processus peut faire sur le système. Cela est défini dans le document Posix.1e (anciennement Posix.6).

Les capacités d'un processus correspondent à des privilèges, aussi les applications courantes ont-elles des ensembles de capacités vides. En dotant un programme d'un jeu restreint de privilèges (par exemple pour modifier sa propre priorité d'ordonnancement, on lui accorde une puissance suffisante pour accomplir son travail, tout en évitant tout problème de sécurité qui pourrait survenir si le programme était détourné de son utilisation normale. Ainsi, même si une faille de sécurité existe dans l'application, et si elle est découverte par un utilisateur malintentionné, celui-ci ne pourra exploiter que le privilège accordé au programme et pas d'autres capacités dangereuses réservées habituellement à *root* (par exemple pour insérer un module personnel dans le noyau).

Un processus dispose de trois ensembles de capacités :

- L'ensemble des capacités **effectives** est celui qui est utilisé à un instant donné pour vérifier les autorisations du processus. Cet ensemble joue un rôle similaire à celui de l'UID effectif, qui n'est pas nécessairement égal à l'UID réel, mais est utilisé pour les permissions d'accès aux fichiers.

- L'ensemble des capacités **transmissibles** est celui qui sera hérité lors d'un appel-système `exec()`. Notons que l'appel `fork()` ne modifie pas les ensembles de capacités ; le fils a les mêmes privilèges que son père.
- L'ensemble des capacités **possibles** est une réserve de privilèges. Un processus peut copier une capacité depuis cet ensemble vers n'importe lequel des deux autres. C'est en fait cet ensemble qui représente la véritable limite des possibilités d'une application.

Une application a le droit de réaliser les opérations suivantes sur ses capacités :

- On peut mettre dans l'ensemble effectif ou l'ensemble transmissible n'importe quelle capacité.
- On peut supprimer une capacité de n'importe quel ensemble.

Un fichier exécutable dispose également en théorie des mêmes trois ensembles. Toutefois, les systèmes de fichier actuels ne permettent pas encore le support pour toutes ces données. Aussi un fichier exécutable Set-UID *root* est-il automatiquement lancé avec ses ensembles de capacités effectives et possibles remplis. Un fichier exécutable normal démarre avec des ensembles effectif et possible égaux à l'ensemble transmissible du processus qui l'a lancé. Dans tous les cas, l'ensemble transmissible n'est pas modifié durant l'appel-système `exec()`.

Les capacités présentes dans le noyau Linux sont définies dans `<linux/capability.h>`. En voici une description, les astérisques signalant les capacités mentionnées dans le document Posix.1e.

Nom	Signification
`CAP_CHOWN (*)`	Possibilité de modifier le propriétaire ou le groupe d'un fichier.
`CAP_DAC_OVERRIDE (*)`	Accès complet sur tous les fichiers et les répertoires.
`CAP_DAC_READ_SEARCH (*)`	Accès en lecture ou exécution sur tous les fichiers et répertoires.
`CAP_FOWNER (*)`	Possibilité d'agir à notre gré sur un fichier ne nous appartenant pas, sauf pour les cas où `CAP_FSETID` est nécessaire.
`CAP_FSETID (*)`	Possibilité de modifier les bits Set-UID ou Set-GID d'un fichier ne nous appartenant pas.
`CAP_IPC_LOCK`	Autorisation de verrouiller des segments de mémoire partagée et de bloquer des pages en mémoire avec `mlock()`.
`CAP_IPC_OWNER`	Accès aux communications entre processus sans passer par les autorisations d'accès.
`CAP_KILL (*)`	Possibilité d'envoyer un signal à un processus ne nous appartenant pas.
`CAP_LINUX_IMMUTABLE`	Modification d'attributs spéciaux des fichiers.
`CAP_NET_ADMIN`	Possibilité d'effectuer de nombreuses tâches administratives concernant le réseau, les interfaces, les tables de routage, etc.
`CAP_NET_BIND_SERVICE`	Autorisation d'accéder à un port privilégié sur le réseau (numéro de port inférieur à 1 024).
`CAP_NET_BROADCAST`	Autorisation d'émettre des données en *broadcast* et de s'inscrire à un groupe *multicast*.
`CAP_NET_RAW`	Possibilité d'utiliser des sockets réseau de type *raw*.
`CAP_SETGID (*)`	Autorisation de manipuler le bit Set-GID et de s'ajouter des groupes supplémentaires.
`CAP_SETPCAP`	Possibilité de transférer nos capacités à un autre processus (dangereux ! ne pas utiliser !).

Nom	Signification
CAP_SETUID (*)	Autorisation de manipuler les bits Set-UID et Set-GID d'un fichier nous appartenant.
CAP_SYS_ADMIN	Possibilité de réaliser de nombreuses opérations de configuration concernant le système proprement dit.
CAP_SYS_BOOT	Autorisation d'arrêter et de redémarrer la machine.
CAP_SYS_CHROOT	Possibilité d'utiliser l'appel-système `chroot()`.
CAP_SYS_MODULE	Autorisation d'insérer ou de retirer des modules de code dans le noyau.
CAP_SYS_NICE	Possibilité de modifier sa priorité d'ordonnancement, ou de basculer en fonctionnement temps-réel.
CAP_SYS_PACCT	Mise en service de la comptabilité des processus.
CAP_SYS_PTRACE	Possibilité de suivre l'exécution de n'importe quel processus.
CAP_SYS_RAWIO	Accès aux ports d'entrée-sortie de la machine.
CAP_SYS_RESOURCE	Possibilité de modifier plusieurs limitations concernant les ressources du système.
CAP_SYS_TIME	Mise à l'heure de l'horloge système.
CAP_SYS_TTY_CONFIG	Autorisation de configurer les consoles.

Lorsque nous examinerons une fonction privilégiée, nous indiquerons quelle capacité est nécessaire pour s'en acquitter. Par contre, nous n'allons pas détailler le moyen de configurer les permissions d'un processus, car l'interface du noyau est sujette aux changements. Il existe depuis Linux 2.2 deux appels-système, **`capset()`** et **`capget()`**, permettant de configurer les ensembles de permissions d'un processus. Toutefois, ils ne sont ni portables ni même garantis d'exister dans les noyaux futurs.

Pour agir sur les privilèges d'une application, il faut employer la bibliothèque `libcap`, qui n'est pas toujours installée dans les distributions courantes. Cette bibliothèque fournit non seulement des fonctions Posix.1e pour modifier les permissions, mais également des utilitaires permettant, par exemple, de lancer une application avec un jeu restreint de privilèges.

On peut trouver la bibliothèque `libcap` à l'adresse suivante :

ftp://linux.kernel.org/pub/linux/libs/security/linux-privs

La segmentation des privilèges habituellement réservés à *root* est une chose très importante pour l'avenir de Linux. Cela permet non seulement à un administrateur de déléguer certaines tâches à des utilisateurs de confiance (par exemple en leur fournissant un shell possédant la capacité `CAP_SYS_BOOT` pour pouvoir arrêter l'ordinateur), mais la sécurité du système est aussi augmentée. Une application ayant besoin de quelques privilèges bien ciblés ne disposera pas de la toute-puissance de *root*. Ainsi, un serveur X11 ayant besoin d'accéder à la mémoire vidéo aura la capacité `CAP_SYS_RAWIO`, mais ne pourra pas aller écrire dans n'importe quel fichier système. De même, un logiciel d'extraction de pistes audio depuis un CD, comme l'application `cdda2wav`, aura le privilège `CAP_SYS_NICE` car il lui faudra passer sur un ordonnancement temps-réel, mais il n'aura pas d'autres autorisations particulières.

Si un pirate découvre une faille de sécurité lui permettant de faire exécuter le code de son choix sous l'UID effectif de l'application – comme nous le verrons dans le chapitre 10 à propos de la fonction `gets()` –, il n'aura toutefois que le privilège du processus initial. Dans les deux exemples indiqués ci-dessus, il pourra perturber l'affichage grâce à l'accès à la

mémoire vidéo, ou bloquer le système en faisant boucler un processus de haute priorité temps-réel. Dans un cas comme dans l'autre, cela ne présente aucun intérêt pour lui. Il ne pourra modifier aucun fichier système (pas d'ajout d'utilisateur, par exemple) ni agir sur le réseau pour se dissimuler en préparant l'attaque d'un autre système. Ses possibilités sont largement restreintes.

Conclusion

Dans ce chapitre, nous avons essayé de définir la notion de processus, la manière d'en créer, et les différents identifiants qui peuvent y être attachés. Une application classique n'a pas souvent l'occasion de manipuler ses UID, GID, etc. Cela devient indispensable toutefois si l'accès à des ressources privilégiées qui doivent être offertes à tout utilisateur est nécessaire. L'application doit savoir perdre temporairement ses privilèges, quitte à les récupérer ultérieurement. De même, certains programmes ayant un dialogue important avec leurs descendants seront amenés à gérer des groupes de processus. Bien entendu, tout ceci est également nécessaire lors de la création de processus démons, comme nous le verrons dans la partie consacrée à la programmation réseau.

Une présentation détaillée des permissions associées aux processus se trouve dans [BACH 1989] *Conception du système Unix*. Nous avons également abordé les principes des capacités Posix.1e, introduites dans Linux 2.2, et qui permettent d'améliorer la sécurité d'une application nécessitant des privilèges. Il faut toutefois être conscient que l'implémentation actuelle de ces capacités est loin d'être aussi riche que ce que propose Posix.1e.