LINUX
Administration

Quatrième édition – Noyau 2.6

Jean-Francois Bouchaudy
Gilles Goubet

Tsoft EDITEUR

EYROLLES

- *Concept de processus*
- *PID, signal*
- *Démon, groupes de processus*
- *ps, kill, fuser, su*
- *Le service cron*

7

La gestion des processus

Objectifs

A l'issue de ce module, le lecteur connaît les concepts fondamentaux des processus du système Linux. Il sait identifier les démons, maîtrise les commandes usuelles et le service `cron` pour automatiser l'exécution de tâches périodiques. Il sait ce que sont les IPC et il est capable de les paramétrer.

Contenu

La gestion des applications
Les processus
L'environnement
Panorama des commandes de gestion des processus
`crontab`
Les bibliothèques dynamiques
Les IPC
Le service `Syslog`

Références

Livre : *The design of the UNIX Operating System,* par M. Bach.
Programmation Linux 2.0, par R. Card, E. Dumas et F. Mével.

La gestion des applications

- **Une application = ensemble de processus**
- **Un processus = une ou plusieurs « threads »**
- **Un processus =**
  - Un fichier exécutable
  - zéro ou plusieurs bibliothèques partagées (.so)
- **Les sources C/C++ assurent la portabilité**
- **Le « man » dit tout**
- **Il faut lire les « logs » !**
- **Activation d'une application : sh, init, inetd et cron**
- **Une application = un « package »**

Applications, processus et threads

Une application Linux en cours d'exécution est composée d'un ou plusieurs processus. Chaque processus correspond à un programme qui s'exécute dans son propre espace d'adressage (espace virtuel), parallèlement aux autres processus. Le noyau d'un système Linux a pour rôle essentiel d'exécuter des processus et de leur permettre de dialoguer.

Un processus peut être composé d'une ou plusieurs thread ou unité d'exécution, ou encore processus léger (Light Wheight Processus ou LWP). Les threads d'un processus s'exécutent en parallèle. L'utilisation de threads constitue une alternative à l'exécution concurrente de processus. Une application complexe peut, bien sûr, utiliser les deux approches.

Processus, langage C et bibliothèques

Chaque processus correspond à un fichier exécutable. Ce fichier est le plus souvent le résultat de la compilation d'un ou de plusieurs fichiers source en langage C/C++.

L'exécutable utilise des bibliothèques. Celles-ci peuvent être incluses dès la compilation (édition de liens statique) ou chargées dynamiquement et partagées avec d'autres exécutables (édition de liens dynamique). Les bibliothèques dynamiques se présentent sous forme de fichiers ayant l'extension « .so » et sont bien sûr nécessaires à l'exécution du logiciel.

Pour automatiser la compilation, on fait habituellement appel à la commande `make`. Cet outil de développement se base sur un fichier *Makefile* qui décrit comment créer le logiciel.

Processus, API Unix/Linux et logiciels libres

Les applications UNIX écrites en langage C accèdent aux ressources système (processus, fichiers et périphériques) grâce à l'API Unix. Cette API est une bibliothèque de fonctions qui permet de réaliser les appels système. C'est sur elle que repose la portabilité d'UNIX : pour porter une application, il suffit de recompiler ses sources C sur la plate-forme de destination. Les deux primitives les plus importantes sont fork(2) et exec(2) qui régissent la création de processus.

Quand on achète une application, on obtient essentiellement des fichiers binaires. La société éditrice possède les sources de l'application et elle est la seule à pouvoir la porter sur différentes plates-formes.

Dans le monde UNIX, le logiciel libre a une place de choix. UNIX lui-même et de nombreux logiciels qu'il intègre sont intimement liés aux logiciels libres. La plupart des logiciels libres ont d'abord été créés pour UNIX. Ils se présentent bien évidemment sous forme de fichiers sources en langage C/C++ pour pouvoir s'adapter à n'importe quelle plate-forme.

Le système Linux intègre l'API UNIX, c'est pourquoi les logiciels conçus pour UNIX fonctionnent sous Linux après recompilation.

La documentation type d'une application Linux

Une application Linux bien documentée est fournie avec des pages de manuel qui la décrivent. Ainsi, la documentation d'applications serveur comme `inetd`, `named` ou `httpd` contient les informations suivantes :

- Les arguments de la ligne de commande.
- Les variables d'environnement qui modifient le comportement de l'application.
- Les signaux auxquels réagit l'application.
- Les fichiers associés à l'application : fichiers de configuration, fichiers logs, fichiers temporaires, etc...
- Des renvois (see also) à d'autres pages de manuel.

Les fichiers de configuration sont généralement en ASCII et abondamment commentés, notons comme exemple le fichier `httpd.conf` d'Apache.

Les journaux de bord (logs)

La manière la plus simple de déboguer une application est de consulter ses journaux de bord, ses « logs ».

La documentation décrit comment les activer et comment ajuster le niveau de commentaire. Elle indique également le nom des fichiers et précise le lien avec le service `syslog`.

`Syslog` permet d'aiguiller les journaux de bord, aussi bien ceux du système que ceux des applications. On peut paramétrer `syslog` pour que l'ensemble des informations journalisées soient réunies dans le fichier */var/log/messages*.

Activation d'une application via un RC

Une application complexe, comme le serveur NFS, est composée de plusieurs processus dont l'activation comporte une liste d'arguments. Le démarrage de l'application nécessite en outre de vérifier l'existence de fichiers et de répertoires. La

gestion de l'application est simplifiée si son démarrage, son arrêt ou son redémarrage sont réalisés par un script unique.

Ce script peut répondre aux conventions de gestion d'application du système et résider dans le répertoire */etc/init.d*. Les liens établis dans les répertoires associés aux niveaux d'init permettent d'automatiser le démarrage et l'arrêt automatique de l'application en même temps que celui du système *(cf. Module 9 : L'arrêt démarrage)*. Un tel script est surnommé RC (Run Command).

Il est bien sûr possible de continuer de gérer l'application manuellement :

```
# cd /etc/init.d
# ./nfs   start        # active le serveur  NFS
# ./nfs   stop      # arrêt du serveur NFS
```

Activation d'une application via inetd

Un serveur réseau peut être activé par un script RC ou par le démon `inetd` *(cf .Module 10 : Le réseau)*. C'est le fichier */etc/inetd.conf* qui décrit alors le démarrage de l'application.

Activation d'une application via un crontab

Une application peut également être exécutée périodiquement grâce à un fichier crontab, pris en compte par le démon `cron`.

Surveillance d'une application

La commande `ps` est le principal outil de surveillance d'une application :

```
# ps -ef  | grep httpd
```

On peut également surveiller les journaux de bord en temps réel :

```
# tail -f  access_log
```

Installation/désinstallation d'une application

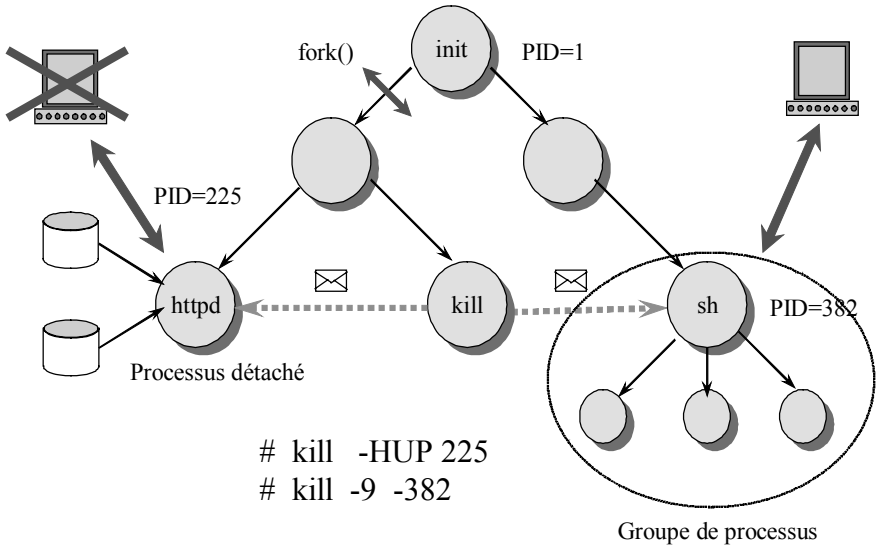
Une application Unix se présente traditionnellement sous forme d'un fichier *TAR* compressé, un « *tarball* ».

Tous les systèmes Linux proposent un outil de gestion de paquetages (packages) qui simplifie l'installation et la suppression des applications. Les principaux sont le système RedHat (RPM) et le système Debian. Ces outils et les formats des paquetages sont malheureusement propres à chaque système.

Les logiciels de gestion de paquetages sont également utilisés pour réaliser les mises à jour de son système. Ce point est très important, autant d'un point de vue de l'exploitation courante que d'un point de vue de la sécurité.

L'installation de logiciel est traitée dans le module suivant.

Les processus



Introduction

Lorsqu'un utilisateur demande à exécuter une commande externe (i.e. `/bin/date`), le noyau Linux crée un processus, une tâche, qui exécute les instructions de la commande. Ce processus est une entité qui, de sa création à sa mort, est identifiée par une valeur numérique, le PID (Process IDentifier) et transite par différents états selon qu'il s'exécute (actif), attend que le noyau lui alloue le processeur (prêt) ou qu'un événement se produise (en attente).

A un instant donné, une valeur de PID est unique dans le système et cela garantit qu'un processus est identifié sans ambiguïté par son PID, alors que le nom de commande ne garantit pas l'unicité. Plusieurs instances de la commande `sh` s'exécutent en même temps quand plusieurs utilisateurs sont connectés. A chaque instance correspond un processus avec un PID unique.

Un processus est associé à un compte utilisateur représenté par son UID. Il est associé également à un ou plusieurs GID. De ces associations découlent les droits que possède le processus.

La création d'un processus est réalisée par la primitive `fork`(2) du noyau. Rappelons que le chiffre 2, situé entre parenthèses, désigne la section 2 du manuel de référence, c'est-à-dire l'ensemble des primitives programmables, entre autres, en langage C. Cette primitive crée un clone du processus qui l'exécute. Le clone, que l'on appelle processus fils, exécute le code d'une autre commande grâce à la primitive `exec`(2). Il en résulte un arbre de processus. Chez un processus fils, Le PPID indique la valeur du PID du processus père. L'ancêtre de tous les processus, `init`, est créé au démarrage du système et se trouve à la racine de cet arbre. Son PID est 1. En paramètre de la primitive `exec`, le code de l'application père peut fournir une liste d'arguments. C'est ce mécanisme qu'utilise le shell pour transmettre les arguments aux processus qu'il active, ceux-là même que l'utilisateur a mis derrière le nom de la commande.

Il existe plusieurs manières d'exécuter un processus :

- En avant-plan (foreground) dans une session de travail, c'est le mode normal d'exécution d'une commande. Le shell de connexion est le père du processus qui exécute la commande. Il attend que le processus fils se termine pour afficher à nouveau l'invite (prompt).
- A l'arrière-plan (background) dans une session de travail, c'est le mode utilisé quand on termine une ligne de commande par « & ». Après avoir créé le processus fils qui exécute la commande, le shell affiche immédiatement l'invite sans attendre la fin du processus fils. Rappelons qu'un processus d'arrière-plan est insensible au caractère de contrôle `CTRL C` qui, normalement, met fin au processus d'avant-plan.
- En mode détaché, un processus détaché n'a plus de terminal de contrôle. On voit le caractère « ? » dans la colonne TTY produite par la commande `ps`.
- Les démons (daemon) désignent communément les processus détachés associés aux services du système Linux, souvent en mode client-serveur. Ils sont créés par les scripts de démarrage, qui contiennent la ligne de commande pour les exécuter, et détruits lors de l'arrêt du système.

Nous venons de décrire les principaux attributs d'un processus : PID, PPID, UID, GID(s), nom de l'exécutable, arguments, TTY et état (actif, endormi, zombie). La commande `ps -ef` affiche la plupart de ces attributs. Il en existe d'autres : le PGID, la réaction aux signaux (*cf. paragraphe suivant*), l'environnement (*cf. chapitre suivant*), la priorité absolue, la priorité temps partagée (NICE), etc. L'ensemble des attributs est décrit dans l'ouvrage de Rémy Card & all: *Programmation Linux 2.O*.

Les signaux

Le seul moyen d'agir sur l'exécution d'un processus détaché est de lui envoyer un signal (*cf. UNIX shell : Module 12*).

Le signal 1 est conventionnellement utilisé pour dire à un démon de relire ses fichiers de configuration.

Le signal 15 (TERM) est conventionnellement utilisé pour dire à un démon de se terminer. C'est d'ailleurs le signal que la commande `shutdown` envoie aux processus actifs.

Le signal 9 tue le processus sans qu'il puisse intercepter le signal et exécuter une quelconque action avant de se terminer. Le signal 9 ne doit, pour cette raison, être utilisé qu'en dernier recours. Les fichiers de configuration et de données du service associé au processus qui est tué brutalement sont certainement dans un état incohérent. Le rétablissement de la cohérence doit alors être réalisé manuellement par l'administrateur et peut s'avérer fort complexe.

Il arrive qu'un processus et ses descendants forment un groupe de processus. Le groupe de processus est identifié par un numéro de groupe, le PGID. Ce numéro de groupe est le PID du processus « leader ». Il est possible d'envoyer un signal à l'ensemble des membres du groupe, y compris le leader (*cf. Panorama des commandes de gestion de processus*).

Quand un processus crée un processus fils, il peut indiquer au noyau qu'il ne souhaite pas être averti de sa terminaison. A défaut, le noyau considère que le processus père souhaite savoir la cause de la mort de son fils, ce qui peut se produire parce qu'il a exécuté la primitive `exit`(2) ou qu'il a reçu un signal ayant provoqué sa mort. Si le fils vient à se terminer avant que le père ne se soit mis en attente de sa fin, le noyau conserve le descripteur du processus pour ensuite pouvoir rendre compte au père. Durant toute cette période, le processus est dit « zombie ». Dans la liste des processus

produite par la commande ps, il apparaît comme <defunct>. Au cas où son père meurt aussi, c'est le processus init qui hérite de la paternité.

Remarque

Un processus zombie compte pour un processus, bien qu'il n'existe plus réellement. En cas de bogue dans une application exécutée par root, leur multiplication peut entraîner l'effondrement du système dès que l'on a atteint le nombre maximum de processus autorisés.

Exemple

Dans l'exemple qui suit, on crée l'application a.out à partir du source C zombie.c, que l'on compile avec gcc (le compilateur du GNU). L'application utilise les principaux appels système : fork(), exec(), wait(), exit(). Le processus initial commence par se dupliquer grâce à la primitive fork(). Le processus fils exécute ensuite, via la primitive exec(), le code de la commande cal avec, comme arguments, ceux donnés à l'appel de a.out.

Lors de la première exécution de a.out, on donne des arguments valides à la commande cal qui se termine correctement et renvoie le code 0. Comme a.out est lancé en tâche de fond, on peut visualiser les processus. On constate que le processus fils, celui qui a exécuté cal, a généré un processus zombie (<defunct>). Il est impossible de le tuer, y compris avec le signal 9. C'est la récupération de son code retour par son père vingt secondes plus tard qui le fait disparaître.

Dans la deuxième exécution de a.out, on tue l'application père grâce à un CTRL-C, ce qui entraîne également la disparition du zombie. En effet, le zombie n'existe que pour transmettre son code retour.

Dans la troisième exécution de a.out, on fournit des arguments incorrects à la commande cal qui retourne le code 1.

```
$ cat zombie.c
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
main(int argc, char ** argv) {
        int status;
        if ( fork() == 0 )
                execlp("cal", "cal", argv[1], argv[2], 0 );
        else {
                sleep(20);
                wait(&status);
                printf("code retour:%d\n", WEXITSTATUS(status));
                exit(0);
        }
}
$ gcc zombie.c
$ a.out  6 1944 &
   juin 1944
 S  M Tu  W Th  F  S
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

4575
$ ps -u jf
```

```
   PID TTY      TIME CMD
  4575 pts/0    0:00 a.out
  4513 pts/0    0:00 sh
  4576          0:00 <defunct>
$ kill -9 4576
$ ps -u jf
   PID TTY      TIME CMD
  4575 pts/0    0:00 a.out
  4513 pts/0    0:00 sh
  4576          0:00 <defunct>
$ code retour:0
$ ps -u jf
   PID TTY      TIME CMD
  4513 pts/0    0:00 sh
$ a.out 9 1732
   septembre 1732
 S  M Tu  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

^C
$ ps -u jf
   PID TTY      TIME CMD
  4513 pts/0    0:00 sh
$ a.out  13  2001
cal : mois incorrect
syntaxe : cal [ [mois] année ]
code retour:1
$
```

Références

Man

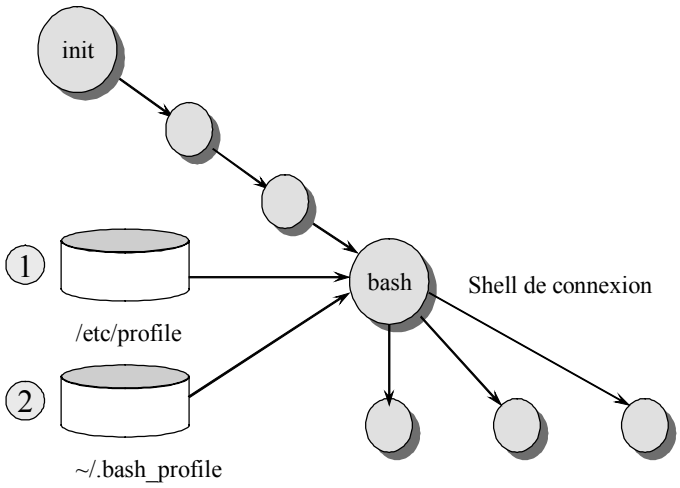
fork(2), exec(2)

Livre

Programmation Linux 2.0, par M. Bach.

The design of UNIX Operating System, par R. Card, E. Dumas et F. Mével.

L'environnement



TERM=vt100; export TERM

Quand un processus crée un processus fils, ce dernier hérite de son père d'un ensemble d'éléments définis par ses ancêtres. Parmi ces éléments, ceux qui intéressent particulièrement l'administrateur sont :

- Le répertoire courant de son père (le répertoire « **.** »).
- Le `umask` qui, rappelons-le, sert à définir les droits par défaut des fichiers créés par le processus.
- Le `ulimit` qui fixe la taille du plus gros fichier que le processus peut créer.
- Les variables d'environnement dont le processus reçoit une copie. Elles ont été créées par un ancêtre du processus, du processus `init` jusqu'à son père (*cf. UNIX shell : Module 5, les variables*).
- Le répertoire racine (« / ») peut être modifié par la commande `chroot`. Le processus et ses descendants ne voient plus que l'arbre issu du nouveau répertoire racine.

Dans la pratique, l'administrateur est amené à ajouter, pour le bon déroulement de logiciels applicatifs, la définition de nouvelles variables d'environnement. Pour cela, il dispose de plusieurs fichiers (*cf. Annexe D : le shell bash, les fichiers de configuration*) :

- Le fichier */etc/profile* pour définir les variables valables pour tous les utilisateurs.
- Les fichiers *~/.bash_profile* ou *~/.profile* des seuls utilisateurs concernés par les variables.

La variable d'environnement la plus célèbre est sans doute la variable TERM qui permet à l'éditeur de texte vi de s'adapter au terminal de l'utilisateur.

ulimit

La commande `ulimit` et le paramètre ULIMIT sont bien connus pour limiter la taille du plus gros fichier qu'un utilisateur peut créer. La commande `ulimit` est une commande interne des shells qui permet de fixer les limites de taille ou d'utilisation de nombreuses ressources système. Le tableau qui suit rappelle quelques options de la commande `ulimit` du `bash`.

Option	*Description*
-a	Visualise toutes les limites.
-f NB	Définit le nombre de blocs de 1 ko écrits autorisés.
-s Nb	Définit la taille de la pile en kilo-octets.
-n Nb	Définit le nombre de descripteurs de fichiers plus 1.
-t Nb	Définit le nombre de secondes de temps CPU pour chaque process.

```
# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 512
virtual memory          (kbytes, -v) unlimited
# ulimit -f 100
# dd if=/dev/zero of=/tmp/fichier bs=1k count=1000
File size limit exceeded
# ulimit -f unlimited
# dd if=/dev/zero of=/tmp/fichier bs=1k count=1000
1000+0 records in
1000+0 records out
# ulimit -t 1
# awk 'BEGIN { var=2.5;while(1){var*=3;var/=3}}' < /dev/null
Killed
```

La variable d'environnement TZ

Les dates et les heures mémorisées sur disque, comme celles de dernière modification d'un fichier, sont exprimées en temps universel ou UTC (« *Universal Time Coordinated* »). Par contre, les dates et les heures affichées par la commande `date` ou bien par la commande `ls`, sont exprimées en heure locale. Le décalage horaire, qui tient compte éventuellement de l'heure d'été, est calculé grâce à la variable d'environnement TZ (*Time Zone*). Cette variable est renseignée lors de l'installation du système et elle est mémorisée dans le fichier locale-archive. Elle n'a donc pas

besoin d'apparaître dans l'espace d'environnement, mais si on l'a définit, la nouvelle valeur est prioritaire pour le processus.

La variable TZ peut être exprimée de deux manières. La première contient au minimum une abréviation associée au fuseau horaire et, éventuellement, la description complète du décalage, y compris celui associé à l'heure d'été. Dans la deuxième forme, plus récente, elle contient le continent et une ville de référence. Cette indication renvoie à un fichier binaire qui contient toutes les informations pour calculer l'heure locale.

Syntaxe traditionnelle

TZ=standard[HH[:MM[:SS]]][heure_d_été[HH[:MM[:SS]]][,début[/heure],fin[/heure]]]]

Exemples en utilisant l'ancienne syntaxe

TZ=GMT

Il n'y a pas de décalage, on travaille en UTC.

TZ=EST

EST (Eastern Standard Time) correspond à un décalage de -5h, la côte est des USA. Le code suffit, car il est reconnu de manière interne.

TZ=AST9

AST est le code utilisé pour l'Alaska, le décalage est de neuf fuseaux vers l'ouest (heure locale = UTC-9). Le décalage est compté positif vers l'ouest, et négatif vers l'est.

TZ=MET-1METDST

MET (Middle European Time) correspond au décalage pour la France (heure locale = UTC+1). La France utilise une heure d'été et le nom associé est METDST (MET Daylight Saving Time). L'époque de changement d'heure n'est pas précisée, elle se produit, par défaut, à 2h le premier dimanche d'avril et à 2h le dernier dimanche d'octobre. Le passage à l'heure d'été ajoute par défaut une heure.

TZ=EST5EDT4,116/2:00:00,298/2:00:00

Décalage utilisé au New Jersey en 1986. Le décalage est de -5h en hiver et de -4h en été. Le passage à l'heure d'été a lieu le 116ème jour de l'année à 2h, et se termine le 298ème jour de l'année à 2h.

Exemples en utilisant la nouvelle syntaxe

TZ=Continent/Ville

Exemples :

TZ=Europe/Paris

TZ=Europe/Zurich

TZ=Australia/Canberra

Exemples complet

```
# echo $TZ

# date
Tue Feb 17 13:44:28 CET 2004
# export TZ=GMT
# date
Tue Feb 17 12:44:57 GMT 2004
```

```
# export TZ=Europe/Paris
# date
Tue Feb 17 13:45:49 CET 2004
# export TZ=EST
# date
Tue Feb 17 07:46:51 EST 2004
# export TZ=AST9
# date
Tue Feb 17 03:47:10 AST 2004
# export TZ=EST5EDT4,116/2:00:00,298/2:00:00
# date
Tue Feb 17 07:48:05 EST 2004
# export TZ=Australia/Canberra
# date
Tue Feb 17 23:51:12 EST 2004
#
```

Internationalisation d'une session

Chaque utilisateur peut déterminer la langue qui est utilisée pendant sa session de travail. En théorie, le choix peut être différent selon qu'il s'agit des messages, du format de la date…

La configuration est réalisée grâce à des variables d'environnement et celle par défaut résulte du choix qui a été fait pendant l'installation. L'administrateur ou les utilisateurs peuvent bien évidemment modifier leur profil. Une variable contient une chaîne de caractères qui définit le pays et une particularité linguistique, telle que « fr_FR » pour la France, « fr_BE » pour le français parlé en Belgique ou « de_CH » pour l'allemand parlé en Suisse allemanique.

L'utilisation de la valeur C pour la variable LANG garantie un paramétrage ISO. Il implique l'usage de la langue anglaise. Cette configuration est requise dans l'écriture de scripts qui désirent exploiter la sortie des commandes.

Les variables d'environnement sont :

Variable	*Description*
LC_MONETARY	Définit le symbole monétaire.
LC_TIME	Définit le format de l'heure et de la date.
LC_COLLATE	Définit les caractères et les jeux de caractères utilisés dans des opérations de tri (*cf. localedef(1), strcoll(3)*.
LC_NUMERIC	Définit le symbole décimal, le séparateur des milliers.
LC-CTYPE	Définit des informations nécessaires à la classification des caractères : numériques, alphabétiques (*cf. isdigit(3), isalpha(3)*).
LC_ALL	Définit une valeur unique pour toutes les variables précédentes.
LANG	Définit une valeur générique pour l'internationalisation.

Les commandes qui sont programmées pour l'internationalisation examinent les variables dans l'ordre suivant : LC_ALL, LC_* et LANG.

Remarque

La rubrique ENVIRONMENT VARIABLES des commandes indique à quelle(s) variable(s) la commande réagit.

La commande `locale` permet de connaître les valeurs actuelles des variables, les autres choix possibles et les jeux de caractères disponibles.

Option	*Description*
-a	Affiche les choix possibles d'internationalisation.
-m	Affiche les jeux de caractères disponibles.

Exemples :

L'exemple qui suit présente les résultats de la commande `locale`.

```
# locale
LANG=fr_FR.UTF-8
LC_CTYPE="fr_FR.UTF-8"
LC_NUMERIC="fr_FR.UTF-8"
LC_TIME="fr_FR.UTF-8"
LC_COLLATE="fr_FR.UTF-8"
LC_MONETARY="fr_FR.UTF-8"
LC_MESSAGES="fr_FR.UTF-8"
LC_PAPER="fr_FR.UTF-8"
LC_NAME="fr_FR.UTF-8"
LC_ADDRESS="fr_FR.UTF-8"
LC_TELEPHONE="fr_FR.UTF-8"
LC_MEASUREMENT="fr_FR.UTF-8"
LC_IDENTIFICATION="fr_FR.UTF-8"
LC_ALL=
# locale -a
C
POSIX
af_ZA
af_ZA.iso88591
an_ES
an_ES.iso885915
ar_AE
...
# locale -m
ANSI_X3.110-1983
ANSI_X3.4-1968
ARMSCII-8
ASMO_449
BIG5
BIG5-HKSCS
BS_4730
BS_VIEWDATA
CP10007
CP1125
CP1250
...
```

L'exemple qui suit illustre le fonctionnement de la commande `date`.

```
# export LANG=fr_FR
# env |grep LANG
LANG=fr_FR
# date
mar fév 17 14:01:24 CET 2004
# export LANG=C
# date
Tue Feb 17 14:05:24 CET 2004
# unset LANG
# date
Tue Feb 17 14:05:30 CET 2004
# export LC_TIME=de_CH
# date
Die Feb 17 14:17:17 CET 2004
# export LC_ALL=es_AR
# date '+%A'
martes
#
```

Références

Man

locale(1), locale(5), locale(7), localedef(1), setlocale(3), bash(1), env(1), ulimit(3), getrlimit(2)

Panorama des commandes de gestion de processus

- **La commande ps : lister les processus**
  - ps –ef # liste de tous les processus
  - ps –ef | grep sendmail # est-ce que sendmail est actif ?
  - ps aux # afficher les ressources utilisées
  - ps –u oracle # les processus associés à un UID
- **Les autres commandes**
  - kill 465; killall sendmail # tuer des processus
  - lsof # liste des fichiers ouverts
  - strace date # les appels système de la commande date

Introduction

Il existe de nombreuses commandes qui ont trait à la gestion des processus et beaucoup ont déjà été étudiées. Pour les plus communes nous ne mentionnons qu'un exemple pour rappeler la forme de leur utilisation.

Panorama des commandes

ps

La commande `ps` affiche des informations sur les processus en cours. La commande `ps` accepte maintenant la syntaxe issue de UNIX BSD, le choix initial de Linux, ainsi que les options de UNIX System V AT&T. Dans la forme initiale de Linux, l'option est représentée par une lettre. Les options de type AT&T sont, comme de coutume, des lettres précédées du caractère « - ».

En adoptant les options de UNIX System V, Linux est conforme au standard UNIX 98 de l'Open Group. Nous recommandons leur utilisation.

Les principales options de type BSD sont :

u	Affiche le nom du propriétaire.
a	Affiche les processus des autres utilisateurs.
l	Affiche des informations supplémentaires (format long).

Les principales options de type System V sont :

-e	Affiche des informations sur tous les processus.

-f	Affiche des informations supplémentaires, notamment le nom du propriétaire du processus (colonne UID), le PID du père du processus (colonne PPID) et la ligne de commande complète qui l'a exécuté (colonne CMD).
-u util	Affiche des informations sur tous les processus de l'utilisateur « util ».
-g	Affiche le numéro de groupe du processus (colonne PGID).

top

La commande `top` affiche les processus qui consomment le plus de ressources systèmes. Dans les premières lignes, elle affiche des informations globales sur le système (charge, mémoire, nombre de processus, nombre de zombies, …). La commande top est étudiée dans le module traitant des performances.

kill

La commande `kill` envoie un signal à un (des) processus. Le signal 9 tue un processus.

killall

La commande `killall` envoie un signal à tous les processus qui exécutent une commande spécifique. Le signal SIGTERM est envoyé par défaut, comme dans la commande `kill`.

su

La commande `su` permet, entre autres, de faire exécuter un processus avec une autre identité que la sienne, ce qui est intéressant pour root qui n'a pas, quand il exécute l'une des formes de la commande `su`**,** à fournir le mot de passe de l'utilisateur dont il emprunte l'identité. Grâce à l'option « -c », il est possible de n'exécuter qu'une commande, fournie immédiatement sur la même ligne de commande que `id`. Cette forme est particulièrement intéressante dans les scripts.

ident

La commande `ident` affiche des informations sur la version d'une commande. Elle est équivalente à la commande `what` d'Unix System V.

strace

La commande `strace` permet d'exécuter une application et de visualiser les appels système qu'elle exécute.

strings

La commande `strings` affiche les chaînes de caractères ASCII présentes dans les fichiers binaires.

fuser

La commande `fuser` fournit des informations sur les processus qui ont ouvert un fichier, même un disque (*cf. Module 4 : Les systèmes de fichiers*).

lsof

La commande `lsof` (*cf. Module 4 : Les systèmes de fichiers*) permet de lister les fichiers ouverts et de connaître les applications qui y accèdent.

at

La commande `at` exécute des commandes en différé (*cf. Service cron*).

crontab

La commande `crontab` exécute des commandes périodiquement (*cf. Service cron*).

env

La commande `env` visualise l'environnement d'un shell.

ldd

La commande `ldd` liste toutes les bibliothèques partagées (les fichiers **.so*) utilisées par un exécutable (*cf. Chapitre Les bibliothèques dynamiques*).

bash

Le shell est un outil extraordinaire de gestion et de supervision des processus. Voici quelques commandes internes du shell.

&

La commande & exécute un processus en arrière-plan.

wait

La commande `wait` attend que les commandes exécutées à l'arrière-plan soient terminées pour poursuivre l'exécution du shell.

Cela n'a d'intérêt que s'il existe plusieurs commandes d'arrière-plan, au moins deux. C'est sinon le mode de fonctionnement normal du shell.

exec

La commande `exec` remplace le code du shell par le code de la commande. Il n'y a pas de création de processus. Il n'y a pas de retour si la commande `exec` réussit.

. unscript

Le shell courant exécute lui-même le script `unscript`.

Ce qu'il faut savoir faire

Activer une application

Pour activer une application simple, il suffit d'exécuter la ou les commandes associées.

```
# sendmail -bd
```

Dans le cas d'une application complexe, il faut utiliser un script RC pour activer et arrêter l'application.

```
# /etc/init.d/sendmail start
Démarrage de sendmail :                                    [  OK  ]
Démarrage de sm-client :                                   [  OK  ]
```

Est-ce qu'une application particulière fonctionne ?

Dans l'exemple suivant, on vérifie avec la commande `ps`, que l'application `sendmail` fonctionne.

```
# ps -ef |grep sendmail
root      1155     1  0 Feb13 ?        00:00:00 sendmail: accepting
connections
smmsp     1163     1  0 Feb13 ?        00:00:00 sendmail: Queue
runner@01:00:00 for /var/spool/clientmqueue
root     10288 10194  0 20:24 pts/1    00:00:00 grep sendmail
#
```

Arrêter une application

Dans la plupart des cas, il suffit grâce à la commande `kill`, d'arrêter les processus qui composent l'application. Si ils portent tous le même nom, la commande `killall` est pratique.

```
# kill 1155 1163
# ps -ef |grep sendmail
root     10290 10194  0 20:29 pts/1    00:00:00 grep sendmail

# killall sendmail
#
```

Si l'application est contrôlée par un script RC, il est plus propre de l'utiliser pour démarrer et arrêter l'application.

```
# /etc/init.d/sendmail stop
Arrêt de sendmail :                                        [  OK  ]
Arrêt de sm-client :                                       [  OK  ]
#
```

Quelle est l'application qui accède en ce moment à un fichier ?

Dans l'exemple suivant, on recherche l'application qui accède au fichier mqueue.

```
# lsof |grep mqueue
sendmail  10486    root  cwd    DIR        3,8     4096    243577
/var/spool/mqueue
#
```

Quelles sont les fichiers actuellement ouverts par une application ?

```
# lsof |grep syslogd
...
syslogd     957    root    2w   REG        3,8     1573    101319
/var/log/messages
syslogd     957    root    3w   REG        3,8      388    101320
/var/log/secure
syslogd     957    root    4w   REG        3,8     2252    101321
/var/log/maillog
syslogd     957    root    5w   REG        3,8     5097    101324
/var/log/cron
syslogd     957    root    6w   REG        3,8        0    101322
/var/log/spooler
syslogd     957    root    7w   REG        3,8      710    101323
/var/log/boot.log
#
```

Pour aller plus loin

La commande ps

```
# ps  # les processus associés au terminal courant
  PID TTY          TIME CMD
 9823 pts/1    00:00:00 bash
10194 pts/1    00:00:00 bash
10521 pts/1    00:00:00 ps
# ps -ef |more   # tous les processus et leurs principaux attributs
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Feb13 ?        00:00:05 init [3]
root         2     1  0 Feb13 ?        00:00:00 [ksoftirqd/0]
...
root       957     1  0 Feb13 ?        00:00:00 syslogd -m 0
root       961     1  0 Feb13 ?        00:00:00 klogd -x
rpc        987     1  0 Feb13 ?        00:00:00 portmap
rpcuser   1007     1  0 Feb13 ?        00:00:00 rpc.statd
root      1088     1  0 Feb13 ?        00:00:00 /usr/sbin/smartd
root      1101     1  0 Feb13 ?        00:00:02 /usr/sbin/sshd
root      1137     1  0 Feb13 ?        00:00:00 xinetd -stayalive -
pidfile /var/run/xinetd.pid
root      1173     1  0 Feb13 ?        00:00:00 gpm -m /dev/mouse -t
imps2
root      1182     1  0 Feb13 ?        00:00:00 crond
daemon    1199     1  0 Feb13 ?        00:00:00 /usr/sbin/atd
# ps -u daemon  # les processus activé par un UID
  PID TTY          TIME CMD
 1199 ?        00:00:00 atd
# ps aux |more # afficher l'usage des ressources
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME
COMMAND
root         1  0.0  0.6  1508  412 ?        S    Feb13   0:05 init
[3]
...
root       957  0.0  0.9  1572  568 ?        S    Feb13   0:00
syslogd -m 0
root       961  0.0  0.5  1516  368 ?        S    Feb13   0:00 klogd
-x
rpc        987  0.0  0.9  1648  560 ?        S    Feb13   0:00
portmap
rpcuser   1007  0.0  1.1  1652  704 ?        S    Feb13   0:00
rpc.statd
root      1088  0.0  1.0  1824  652 ?        S    Feb13   0:00
/usr/sbin/smartd
root      1101  0.0  2.2  3680 1396 ?        S    Feb13   0:02
/usr/sbin/sshd
root      1137  0.0  1.4  2132  888 ?        S    Feb13   0:00
xinetd -stayalive
 -pidfile /var/run/xinetd.pid
root      1173  0.0  0.6  1708  396 ?        S    Feb13   0:00 gpm -
m /dev/mouse
 -t imps2
root      1182  0.0  0.9  1556  592 ?        S    Feb13   0:00 crond
# ps -o pid,ppid,pgid,cmd  # afficher le PID, le PPID et le PGID
  PID  PPID  PGID CMD
```

```
 9823  9821  9823 -bash
10194  9823 10194 bash
10535 10194 10535 ps -o pid,ppid,pgid,cmd
#
```

La commande kill

```
# kill -9 1540      # tue le processus de PID 1540.
# kill -l           # affiche la liste des signaux.
# kill -2  0        # envoie le signal à tous les processus du
groupe du shell.
# kill -2 -1        # envoie le signal à tous les processus dont
l'UID réel est identique à l'UID effectif du  propriétaire du shell
émetteur.
# kill -2 -1540   # envoie le signal à tous les processus du groupe
1540.
```

La commande fuser

```
# fuser /var/spool/mqueue
/var/spool/mqueue:   10486c
#
```

La commande su

```
# su pierre # prendre l'identité de pierre.
$ exit
#
# su  -  pierre   # prendre l'identité de pierre et changer
d'environnement, le fichier   ~/ .bash_profile de pierre est
exécuté.
$ exit
# su pierre -c "id" # exécute la commande avec l'identité de pierre.
uid=200(pierre) gid=51(stage) groups=51(stage),52(formateur)
#
```

La commande ident

```
# ident mail
mail:
     $OpenBSD: version.c,v 1.4 1996/06/08 19:48:46 christos Exp $
     $OpenBSD: aux.c,v 1.4 1996/06/08 19:48:10 christos Exp $
     $OpenBSD: cmd1.c,v 1.5 1996/06/08 19:48:11 christos Exp $
     $OpenBSD: cmd2.c,v 1.5 1996/06/08 19:48:13 christos Exp $
```

La commande strace

La commande `strace` mémorise les appels systèmes. L'appel le plus intéressant du point de vue de l'administrateur est `open`(), d'ouverture de fichier. La commande `strace` permet ainsi de connaître dans le temps, les différents fichiers ouverts par l'application (ou les fichiers que l'application a tenté d'ouvrir).

```
# strace date 2> /tmp/log
Tue Feb 17 21:13:09 CET 2004
# head /tmp/log
execve("/bin/date", ["date"], [/* 23 vars */]) = 0
uname({sys="Linux", node="carapuce.pokemon", ...}) = 0
brk(0)                                  = 0x8053000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
```

```
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=22045, ...}) = 0
old_mmap(NULL, 22045, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40000000
close(3)                                = 0
open("/lib/tls/librt.so.1", O_RDONLY)   = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0p\256[\000"..., 512)
= 512
# grep '^open' /tmp/log
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
open("/lib/tls/librt.so.1", O_RDONLY)   = 3
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
open("/lib/tls/libpthread.so.0", O_RDONLY) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
open("/usr/lib/locale/es/LC_TIME", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("/etc/localtime", O_RDONLY)        = 3
#
```

La commande `strace` peut visualiser les appels d'un daemon et de ses enfants.

```
# ps -e | grep sendmail
10486 ?        00:00:00 sendmail
# strace -p 10486 -f > /tmp/log 2>&1
^C
# more /tmp/log
read(5, "0.11 0.04 0.01 2/38 10581\n", 1024) = 26
close(5)                                = 0
time([1077049402])                      = 1077049402
rt_sigprocmask(SIG_UNBLOCK, [ALRM], [ALRM], 8) = 0
select(5, [4], NULL, NULL, {5, 0})      = 1 (in [4], left {4,
899000})
accept(4, {sa_family=AF_INET, sin_port=htons(1110),
sin_addr=inet_addr("127.0.0.1")}, [16]) = 5
rt_sigprocmask(SIG_BLOCK, [ALRM], [], 8) = 0
pipe([6, 7])                            = 0
...
[pid 10486] select(5, [4], NULL, NULL, {5, 0} <unfinished ...>
[pid 10583] getpid()                    = 10583
[pid 10583] rt_sigprocmask(SIG_UNBLOCK, [ALRM], [ALRM CHLD], 8) = 0
```

La commande strings

```
# strings /bin/date |more
/lib/ld-linux.so.2
PTRh
...
hours
minutes
seconds
file
reference
rfc-822
universal
```

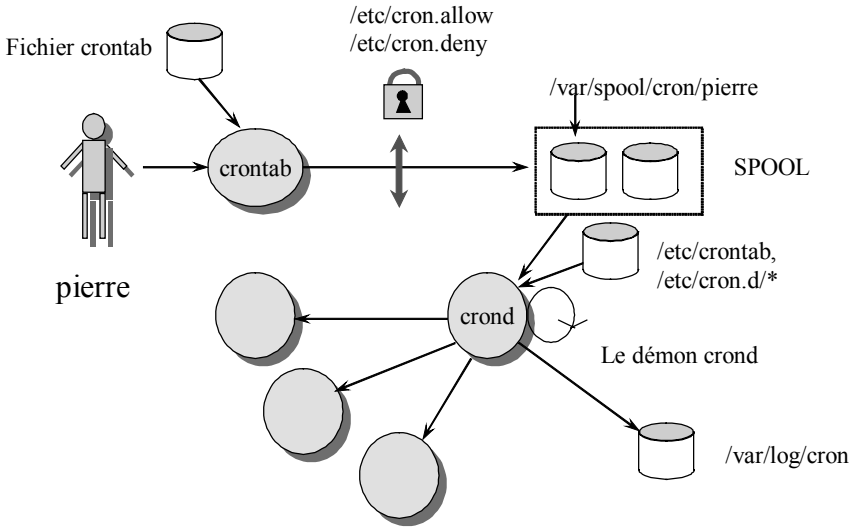
```
help
version
Report bugs to <%s>.
bug-coreutils@gnu.org
invalid date `%s'
standard input
```

Références

Man

ps(1), kill(1), killall(1), su(1), ident(1), strace(1), strings(1), lsof(8), fuser(1), env(1), bash(1)

crontab



Exécution périodique ou différée des commandes

Le démon `crond` exécute des commandes pour des utilisateurs. L'heure et la date d'exécution de ces commandes sont régies par :

- La périodicité (tou(te)s les) quand elles sont soumises par la commande `crontab`.
- Le démon `atd` exécute les commandes différées selon :
- L'échéance (à telle heure) quand elles sont soumises par la commande at.
- A leur tour et dès que possible quand elles sont soumises par la commande `batch`.

Les démons `crond` et `atd` définissent des principes et des contraintes d'utilisation qui sont toujours valables, quel que soit le mode de soumission :

1. Pour soumettre une requête, il faut être un utilisateur autorisé. L'administrateur dispose, pour définir qui est autorisé, des fichiers *cron.allow* et *cron.deny* pour la commande `crontab` et des fichiers *at.allow* et *at.deny* pour les commandes `at` et `batch`. Ces fichiers sont situés dans le répertoire */etc*.
   Comprenons l'utilisation avec *cron.allow* et *cron.deny*. Elle est similaire avec *at.allow* et *at.deny*.
   Si les fichiers n'existent pas, il existe deux cas de figure selon la configuration du système : seul l'administrateur (« root ») est autorisé à exécuter la commande `crontab` ou tous les utilisateurs. Tous les utilisateurs sont autorisés dans le cas de la distribution Red Hat.
   Si le fichier *cron.allow* existe, les utilisateurs dont il contient les noms, un par ligne, sont autorisés à exécuter la commande `crontab`, les autres non.
   Si le fichier *cron.deny* existe, les utilisateurs dont il contient les noms, un par ligne, n'ont pas le droit d'exécuter la commande `crontab`, les autres oui.

   cathy, pierre et root sont les seuls autorisés.

```
# cd /etc
# more cron.allow
root
pierre
cathy
```

Tout le monde est autorisé.

```
# ls cron.*
cron.deny
# more cron.deny          # il est vide
#
```

2. `crond` mémorise l'historique des commandes qu'il exécute dans le fichier */var/log/cron* qu'il faut purger périodiquement.
3. Lorsque le démon `crond` exécute une commande, il définit un environnement réduit composé des variables HOME, LOGNAME, SHELL qui est initialisé avec une valeur par défaut */bin/sh* et une variable PATH réduite à */bin:/usr/bin*. Les commandes qui ont besoin d'un environnement plus complet peuvent être appelées dans un script qui crée les autres variables. Les scripts doivent préciser le shell qui les exécute grâce au pseudo-commentaire « # ! » (*cf. UNIX shell : Module 4, Les scripts shell*). Une autre solution consiste à définir les variables d'environnement dans le fichier des requêtes (*cf. crontab(2/2)*).
4. Pour les commandes qui nécessitent une entrée standard, il est nécessaire de prévoir une redirection explicite en entrée.
   A défaut de redirection de la sortie standard et des erreurs, les résultats et les messages d'erreur sont envoyés dans la boîte à lettres.
5. Les répertoires des requêtes des démons `crond` et `atd` sont respectivement */var/spool/cron* pour la commande `crontab` et */var/spool/at* pour les commandes `at` et `batch`.
   Le fichier de requêtes d'un utilisateur de la commande `crontab`, une fois transféré dans le répertoire des requêtes, porte le nom de l'utilisateur. Cela limite à un le nombre de fichiers de requêtes actifs que peut posséder un utilisateur.

Remarques

- La commande `crontab` est très certainement la plus utilisée par l'administrateur à qui cela permet d'automatiser l'exécution de nombreuses commandes, dont les sauvegardes et la surveillance de l'activité du système.
- L'administrateur dispose du fichier */etc/crontab* pour faire exécuter périodiquement des commandes avec l'identité de n'importe quel utilisateur du système.

Un fichier crontab

On a coutume d'appeler un fichier au format de la commande `crontab`, une « cron table ».

L'utilisateur la crée avec un éditeur de texte. Une ligne peut être une ligne de commande (une requête), une ligne de commentaire qui commence alors par le symbole # ou comporter la définition d'une variable d'environnement pour les commandes.

Les variables HOME et SHELL qui indiquent respectivement le répertoire de connexion et le shell qui doit exécuter les commandes peuvent être redéfinies, mais pas la variable LOGAME. Il est possible de désigner l'utilisateur qui doit recevoir les résultats ou les erreurs par courrier en créant la variable MAILTO qui contient alors le nom du récepteur.

Les lignes vides et les espaces en début de ligne sont ignorés.

Une ligne de commande est constituée de six champs, séparés par des espaces ou des tabulations. Les cinq premiers définissent la périodicité et le dernier la commande à exécuter :

Minutes	Heures	Jours du mois	Mois	Jour de la semaine	Commande

Le contenu d'un champ exprimant un élément de la périodicité peut être :

- Une valeur pour indiquer à quel moment il faut exécuter la commande. La valeur 15 dans le champ minute signifie à la quinzième minute, dans le champ jour du mois, le 15 du mois.
- Une liste de valeurs, séparées par des virgules, pour indiquer à quels moments il faut exécuter la commande. La liste 1,3,5,7 dans le champ mois signifie janvier, mars, mai et juillet.
- Un intervalle pour représenter de « tant à tant ». L'intervalle 1-3 dans le champ jour de la semaine signifie du lundi au mercredi.
- Le caractère * pour indiquer le plus grand intervalle possible. * est équivalent à 0-6 dans le champ jour de la semaine.

Plusieurs requêtes sont présentées dans le l'exemple suivant :

1. La commande `date` est exécutée toutes les minutes. Le résultat de chaque exécution s'ajoute au fichier */tmp/log* au lieu d'être envoyé dans la boîte aux lettres.
2. La commande `who` tous les jours à 21 heures.
3. La commande `cal` à 0 heure, le premier du mois seulement si le mois ne débute pas un dimanche.
4. La commande `touch` à 12 heures 30 le 3, 8 et 15 des mois de mars à octobre.

```
$ more moncron
     *      *      *      *     *  date  >> /tmp/log 2>&1
     0     21      *      *     *  who
     0      0      1      *   1-6  cal
    30     12 3,8,15   3-10     *  touch .bash_profile
```

Remarques

- Il est possible de définir une fréquence en faisant suivre l'intervalle par un pas, sous la forme « intervalle/pas ».
  */10 dans le champ des minutes est équivalent à 0,10,20,30,40,50.
  6-20/2 dans le champ heure est équivalent à 6,8,10,12,14,16,18,20.
- On peut indiquer les noms des jours de la semaine et des mois plutôt que leur numéro. Les noms sont définis en saisissant les trois premiers caractères du mot. « sun » signifie dimanche, « mon » lundi …
- Si le champ commande comporte le caractère %, non protégé, tous les caractères qui suivent, dans la ligne, définissent l'entrée standard de la commande. C'est la forme cron des « here document » du shell.
  Envoi du message « Tout le monde debout ! ! » à tous les utilisateurs, chaque jour à huit heures du matin.
  0 8 * * * wall %Tout le monde debout ! !

La commande crontab

Syntaxe

crontab [-u user] file

crontab [-u user] { -l | -r | -e }

Options

-u user Spécifie l'utilisateur, par défaut l'utilisateur courant.

-l Liste le crontab actif de l'utilisateur.

-r Supprime le crontab de l'utilisateur.

-e Edite le crontab de l'utilisateur

Description

La commande `crontab` permet à un utilisateur de créer, modifier, visualiser ou supprimer une table `cron`. L'administrateur peut agir sur une table d'un utilisateur quelconque

Ce qu'il faut savoir faire

La gestion des tables `cron` est essentielle. Voici les opérations minimales que tout administrateur Linux doit savoir faire.

Soumission d'une table cron

Pour soumettre la « cron table » au démon `cron`, l'utilisateur crée un fichier de requête, puis exécute la commande `crontab`.

```
$ vi  moncron   # ou tout autre éditeur
$ more moncron
* * * * * date >> /tmp/log 2>&1
$ crontab moncron
```

Le fichier de requête est copié dans */var/spool/cron* avec pour nom, celui de l'utilisateur.

L'administrateur peut créer un fichier de requêtes en endossant l'identité d'un autre utilisateur.

```
# crontab –u pierre moncron
```

Visualisation des tables cron actives

Pour connaître les requêtes soumises au démon `cron`, c'est-à-dire visualiser la « cron table » active, il faut exécuter la commande :

```
$ crontab –l
no crontab for paul
```

L'administrateur peut visualiser la cron table d'un utilisateur quelconque.

```
# crontab -u pierre -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.10776 installed on Tue Feb 17 22:00:31 2004)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17 03:20:37 vixie
Exp $)
* * * * * date >> /tmp/log 2>&1
```

Les cron table sont mémorisées dans le répertoire */var/spool/cron*. L'administrateur peut donc les voir, les archiver, etc.

```
# cd /var/spool/cron
# ls
cathy  pierre
#
```

Arrêt de l'exécution de la table cron active

Pour arrêter l'exécution de la « cron table » active, il faut exécuter la commande :

```
$ crontab -r
```

Modification de la table cron active

Pour modifier la « cron table » active, il faut exécuter la commande :

```
$ crontab -e
```

L'édition est réalisée via l'éditeur de texte `vi`. La commande `crontab -e` permet de créer directement une cron table active initiale, donc vide.

Le fichier /etc/crontab et le répertoire /etc/cron.d

En plus des fichiers de requêtes du répertoire */var/cron*, le démon `crond` exécute également les requêtes du fichier */etc/crontab* et les requêtes des fichiers présents dans le répertoire */etc/cron.d*. C'est par ce biais que se fait, entre autres, la mise à jour quotidienne de la base de données d'aide simplifiée « whatis ».

Une ligne de ces fichiers comporte les cinq champs qui définissent la périodicité. Le sixième champ indique l'identité du propriétaire du processus qui exécutera la commande. Le septième champ définit la commande à exécuter.

Le fichier */etc/crontab* de base utilise le script */usr/bin/run-parts* pour exécuter :

toutes les commandes du répertoire */etc/cron.hourly* toutes les heures

toutes les commandes du répertoire */etc/cron.daily* toutes les nuits

toutes les commandes du répertoire */etc/cron.weekly* chaque dimanche

toutes les commandes du répertoire */etc/cron.monthly* le premier jour de chaque mois

Pour exécuter une commande avec l'une de ces périodicités, il suffit à l'administrateur de copier la commande dans le répertoire correspondant. Il est sinon possible d'ajouter une ligne au format des requêtes cron, directement dans le fichier */etc/crontab*.

```
# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Le service anacron

Sur un serveur fonctionnant 24 heures sur 24, le service `cron` permet de gérer complètement les tâches périodiques. Dans le cas d'une station qui est allumée et éteinte chaque jour, le service `anacron` est très utile. En effet, le démon `anacron` vérifie que les commandes ont bien été exécutées dans la période indiquée. Si ce n'est pas le cas, le démon active la commande après un délai déterminé.

Le démon `anacron` est normalement activé par le RC */etc/init.d/anacron*.

Le démon enregistre les actions accomplies dans des fichiers du répertoire */var/spool/anacron*.

La configuration du service est réalisée via le fichier */etc/anacrontab*. Chaque ligne de ce fichier comporte les champs suivants :

Période délai identificateur-de-travail commande

La période est exprimée en jours, le délai en minutes. L'identificateur est associé à la commande dans les messages. Le dernier champ est la commande activée par anacron.

```
# more /etc/anacrontab
# /etc/anacrontab: configuration file for anacron

# See anacron(8) and anacrontab(5) for details.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

1       65      cron.daily              run-parts /etc/cron.daily
7       70      cron.weekly             run-parts /etc/cron.weekly
30      75      cron.monthly            run-parts /etc/cron.monthly
#
```

at

La commande `at` permet de faire exécuter des commandes à une heure et une date données. C'est le démon `atd` qui prend en charge l'exécution des requêtes.

La commande `at` lit les commandes à exécuter depuis l'entrée standard ou dans le fichier associé à l'option `-f`. Il est aussi possible de désigner le fichier en redirigeant l'entrée standard.

$ at -f fichier_requête heure [date]

$ at heure [date] < fichier_requête

L'heure et la date sont au format standard d'UNIX (*cf. man date*) ou intègrent des mots clés propres à la commande `date` (*cf. man at*) :

```
# cat fcommande
exec > /tmp/log 2>&1
date
ps -ef
echo FIN
# at 1500 <fcommande    # exécuter le fichier fcommande à 15 heures.
job 1 at 2004-02-17 15:00
# at midnight < fcommande  # exécuter le fichier fcommande à minuit.
job 1 at 2004-02-18 00:00
# at now + 3 hours < fcommande  # exécuter le fichier fcommande
dans 3 heures.
job 1 at 2004-02-17 18:00
```

Pour connaître les travaux soumis, il faut exécuter la commande `at -l` ou la commande `atq` :

```
$ at -l
1  2004-02-17 15:00 a pierre
2  2004-02-18 00:00 a pierre
3 2004-02-17 18:00 a pierre
```

Pour supprimer un travail avant qu'il ne soit exécuté, il faut exécuter la commande `at -d` ou la commande `atrm` :

```
$ at -d 2  # Supprime le travail numéro 2
```

batch

La commande `batch` utilise le même schéma de fonctionnement que `at`, mais on ne précise pas d'heure ni de date d'exécution. Le travail soumis est mis en attente dans une « FIFO » et exécuté à son tour et quand la charge système le permet.

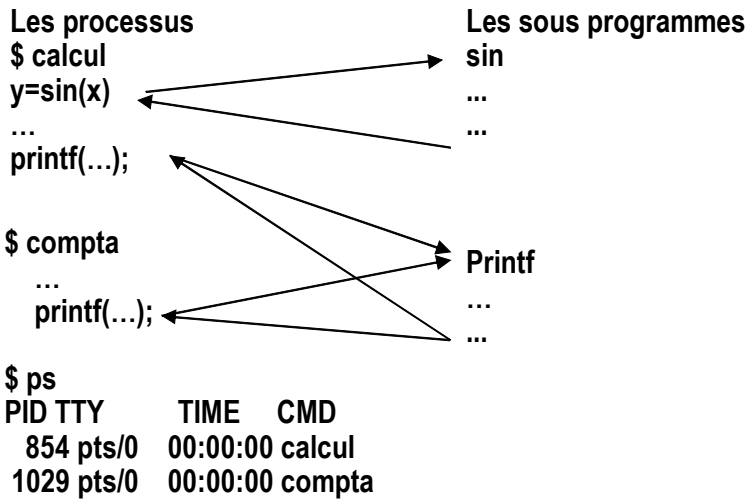
```
$ batch < fcommande
$ at -l     # Visualiser les travaux en attente. Les travaux ont
l'extension « .b »
1       2004-02-17 14:01 a pierre
2       2004-02-18 00:00 a pierre
3       2004-02-17 11:04 b pierre
$ at -d  3         # Supprimer le travail numéro 3
```

Références

Man

crond(8), crontab(1), crontab(5), cron(8), atd(8), at.deny(5), at.allow(5), anacron(8), anacrontab(5)

Les bibliothèques dynamiques



Les bibliothèques dynamiques, le concept

Un programme binaire, sous sa forme exécutable, fait appel à des bibliothèques de sous-programmes. Chaque sous-programme d'une bibliothèque est une unité de programme qui réalise une fonction particulière, comme écrire des données sur la sortie standard, ce que réalise le sous-programme `printf` ou calculer le sinus d'un angle, ce que réalise le sous-programme `sin`. Une bibliothèque est spécialisée. La bibliothèque mathématique contient les fonctions cosinus, sinus, tangente…

Les sous-programmes des bibliothèques peuvent être inclus dans le fichier image de l'exécutable. On parle alors d'édition de lien statique. Le programme est alors autonome. Il suffit de disposer du fichier qui contient le programme pour exécuter l'application. C'est le mode que l'on choisit pour installer un logiciel embarqué sur une machine cible sur laquelle il n'existe rien d'autre que le noyau Linux et le logiciel.

On peut également utiliser des bibliothèques dynamiques (DLL : Dynamic Link Library). Les bibliothèques sont associées au processus pendant l'exécution. On parle d'édition de lien dynamique. Le code des sous-programmes n'est pas inclus dans le fichier image de l'exécutable. Leur présence est nécessaire au moment de l'exécution de l'application. Les bibliothèques sont partagées par tous les exécutables qui y font appel simultanément. C'est pourquoi les termes de bibliothèque dynamique et de bibliothèque partagée (Shared library) sont synonymes.

Le choix de l'édition de lien, statique ou dynamique, se fait au moment de la compilation de l'application. Il nécessite, en conséquence, la possession des sources.

Remarque

La taille d'un fichier exécutable produit avec une édition de lien statique est plus importante que celui obtenu en réalisant une édition de lien dynamique. Le second ne contient pas les sous programmes des bibliothèques

Les bibliothèques

Une bibliothèque se présente sous la forme d'un fichier qui possède l'extension `.a` (« *archive* ») pour une bibliothèque statique et l'extension `.so` (shared object) pour une bibliothèque partagée.

Le nom complet d'une bibliothèque partagée inclut son numéro de version, sous la forme MAJEUR.MINEUR. Le nom *libtermcap.so.2.0.8* désigne la révision de majeur 2 et de mineur 0.8 de la bibliothèque termcap.

Une application peut utiliser une bibliothèque de version plus récente que celle utilisée lors de la compilation, à condition que le majeur reste identique. S'il existe plusieurs applications qui ont été conçues à partir de majeurs différents, cela implique de conserver les différentes versions de ces bibliothèques, une version par majeur.

La politique normalement suivie lors de l'installation d'une application est de créer les bibliothèques dynamiques dans le répertoire */lib* et d'avoir, d'autre part, un lien symbolique pour chaque majeur, qui pointe sur la bibliothèque la plus récente compatible avec ce majeur, par exemple :

```
# ls -l libresolv*
-rwxr-xr-x  1 root   root  169720 fév 29  2000 libresolv-2.1.3.so
lrwxrwxrwx  1 root   root      18 sep  1 12:38 libresolv.so.2 ->
libresolv-2.1.3.so
```

Remarque

Si pour une raison ou pour une autre, des bibliothèques essentielles, comme libc , la bibliothèque standard du langage C, venaient à ne plus être accessibles, les commandes ne fonctionneraient plus. Il faut donc impérativement posséder des disquettes de démarrage qui contiennent les principales commandes (`cp`, `ls`, …), créées par édition de lien statique. Ces disquettes permettent de réparer le système, par exemple, en restaurant les bibliothèques (*cf. Annexe J : Dépannage*).

Pour la raison que l'on vient de mentionner, la mise à jour de ces bibliothèques est périlleuse. Il est plutôt conseillé de réaliser une mise à jour du système. Pour y procéder, il faut utiliser la commande `ln -sf` qui permet de remplacer un lien sans qu'il y ait de vacance.

Emplacement des bibliothèques

Les bibliothèques dynamiques sont chargées par l'utilitaire `ld.so`. Ce chargeur recherche les bibliothèques en respectant l'ordre suivant :

- Les répertoires mentionnés par la variable d'environnement LD_LIBRARY_PATH sont parcourus.
- Le fichier cache */etc/ld.so.cache* est ensuite pris en compte.
- La recherche se termine en scrutant les répertoires */lib* et */usr/lib*.

Le fichier */etc/ld.so.cache*, contient une liste de chemins de bibliothèques. Il est normalement mis à jour au démarrage par la commande `ldconfig`, qui se base sur son fichier de configuration */etc/lib.so.conf*. La commande `ldconfig`, avec l'option `-n`, peut être exécutée à tout moment, pour prendre en compte de nouveaux répertoires.

Compilation et édition de liens

La commande `cc` réalise la compilation et l'édition de liens d'un programme. La compilation consiste dans la production d'un fichier contenant un code binaire non

exécutable et l'édition de lien dans celle d'un fichier binaire exécutable. C'est à l'édition de liens que l'on indique le type des bibliothèques. Par défaut, c'est la bibliothèque dynamique qui est utilisée. L'option `-static` entraîne l'utilisation des bibliothèques statiques.

Les commandes d'information

Plusieurs commandes peuvent donner des renseignements sur les bibliothèques partagées.

La commande `ldd` liste les bibliothèques partagées d'un exécutable. L'ensemble de ces bibliothèques doit être accessibles.

La commande `objdump` liste les en-têtes d'un fichier au format ELF, par exemple un exécutable ou une bibliothèque partagée. L'option `-x` est la plus intéressante du point de vue de l'édition de liens dynamique.

La commande `ldconfig` permet de configurer le cache. L'option `-p` affiche le cache et l'option `-v` scrute les différents répertoires mentionnés dans le fichier */etc/lib.so.conf*.

La commande `ltrace` permet de tracer les appels aux bibliothèques.

La bibliothèque libc

La bibliothèque libc est utilisée par l'ensemble des applications. Son rôle est primordial. En effet, la portabilité des applications sous Linux est autant associée à une version du noyau qu'à la version de la bibliothèque C. L'utilisation d'un logiciel récent nécessite souvent une mise à jour de cette bibliothèque.

Exemples

Compilation statique et dynamique d'une application :

```
$ cat motcode.c
#include <stdio.h>
#include <string.h>
main()
{
 char *motcode="linux"; char motut[100];
 printf("Saisissez un mot : "); gets(motut);
 if ( ! strcmp(motut,motcode))
    printf("Bravo, le mot codé est linux\n");
 else
    printf("Perdu, il fallait trouver linux\n");
}
$ cc -static motcode.c -o motcode.sta    # compilation en statique
/tmp/ccXnFIwl.o(.text+0x35): In function `main':
: warning: the `gets' function is dangerous and should not be used.
$ cc  motcode.c -o motcode.dyn           # compilation en dynamique
/tmp/ccEDft4x.o(.text+0x35): In function `main':
: warning: the `gets' function is dangerous and should not be used.
$ ls -l mot*.??? # motcode.dyn est dynamique, motcode.sta est
statique
-rwxrwxr-x    1 gilles   gilles      12103 nov  7 17:21 motcode.dyn
-rwxrwxr-x    1 gilles   gilles     965445 nov  7 17:22 motcode.sta
$ ldd mot*.???
motcode.dyn:
```

```
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0x00430000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00418000)
motcode.sta:
        not a dynamic executable
```

Visualiser les bibliothèques nécessaires à l'exécution d'une application :

```
$ ldd /bin/tcsh
        libnsl.so.1 => /lib/libnsl.so.1 (0x4001c000)
        libtermcap.so.2 => /lib/libtermcap.so.2 (0x40032000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x40036000)
        libc.so.6 => /lib/libc.so.6 (0x40063000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ objdump -x /bin/tcsh | grep NEEDED
/bin/tcsh: no symbols
  NEEDED      libnsl.so.1
  NEEDED      libtermcap.so.2
  NEEDED      libcrypt.so.1
  NEEDED      libc.so.6
```

Créer une bibliothèque partagée, utiliser la variable LD_LIBRARY_PATH :

```
$ cat main.c
main() {
        int a=3,b=4,c;
        c = add(a,b);
        printf("%d\n", c );
}
$ cat add.c
int add(int x, int y ) {
        return x+y;
}
$ cat mult.c
int mult(int x, int y ) {
        return x*y;
}
$ gcc -c main.c
$ gcc -c add.c
$ gcc -c mult.c
$ gcc -shared -o libz.so add.o mult.o
$ gcc main.o libz.so
$ ./a.out
./a.out: relocation error: ./a.out: undefined symbol: add
$ ldd a.out
        linux-gate.so.1 =>  (0xffffe000)
        libz.so => /usr/lib/libz.so (0x00594000)
        libc.so.6 => /lib/tls/libc.so.6 (0x00430000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00418000)
$ LD_LIBRARY_PATH=`pwd`
$ # ou bien ou copie libz.so dans /usr/lib
$ export LD_LIBRARY_PATH
$ ./a.out
7
$ ldd a.out
        linux-gate.so.1 =>  (0xffffe000)
        libz.so => /home/pierre/libz.so (0x40001000)
        libc.so.6 => /lib/tls/libc.so.6 (0x00430000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00418000)
```

Créer une bibliothèque statique :

```
$ ar r libz.a add.o mult.o
ar: creating libz.a
$ ranlib libz.a
$ gcc -static main.o libz.a
$ ./a.out
7
$ ldd a.out
        not a dynamic executable
$
```

Visualiser les appels à des bibliothèques :

```
$ ltrace date 2> /tmp/log
Wed Feb 18 00:43:49 CET 2004
$ more /tmp/log
__libc_start_main(0x08049540, 1, 0xbffffae4, 0x0804e338, 0x0804e380
<unfinished
...>
getenv("_POSIX2_VERSION")                          = NULL
getenv("POSIXLY_CORRECT")                          = NULL
setlocale(6, "")                                   = NULL
bindtextdomain("coreutils", "/usr/share/locale") =
"/usr/share/locale"
textdomain("coreutils")                            = "coreutils"
__cxa_atexit(0x0804d440, 0, 0, 0x0804e4e6, 0xbffffa58) = 0
getopt_long(1, 0xbffffae4, "Rd:f:r:s:uI::", 0x0804e640, NULL) = -1
dcgettext(0, 0x0804e537, 5, 0x0804e640, 0)         = 0x0804e537
clock_gettime(0, 0xbffff9d8, 0, 0, 5)              = 0
localtime(0xbffff970)                              = 0x00568480
nl_langinfo(131180, 0x0045a5b6, 0x00565998, 0xbffff9d8, 0xbffffae4)
= 0x0055787d
realloc(NULL, 200)                                 = 0x08053ca0
strftime("Wed", 1024, "%a", 0x00568480)            = 3
memcpy(0x08053ca0, "Wed", 3)                       = 0x08053ca0
strftime("Feb", 1024, "%b", 0x00568480)            = 3
memcpy(0x08053ca4, "Feb", 3)                       = 0x08053ca4
memcpy(0x08053ca8, "18", 2)                        = 0x08053ca8
memset(0x08053cab, '0', 1)                         = 0x08053cab
memcpy(0x08053cac, "0", 1)                         = 0x08053cac
```

Références

Man

ldd(1), objdump(1), ltrace(1), ld.so(8), ldconfig(8)

Howto

Program-Library-HOWTO

Les IPC

- **Les différents IPC (« *Inter Process Communication* »)**
  - Les files d'attente de messages inter processus
  - Les segments de mémoire partagés
  - Les tableaux de sémaphores
- **Les commandes**
  - ***ipcs*** Liste les IPC actifs par type d'IPC
  - ***ipcrm*** Permet de détruire une IPC

Introduction

Les IPC (Inter Process Communication) désignent des mécanismes de synchronisation et de communication implémentés dans le noyau Linux, auxquels les processus applicatifs font appel pour échanger des données ou régler des problèmes de concurrence d'accès à des ressources.

Les processus utilisent, pour cela, des primitives du noyau. Les IPC sont décrits dans la section 2 du manuel de référence.

En quoi, alors, l'administrateur est-il concerné ?

Cela le concerne à trois niveaux :

- Parfaire sa connaissance globale du système.
- Etre capable d'identifier les IPC utilisés par les processus, vérifier que leur nombre est suffisant et à en supprimer à la main en cas de « plantage » d'un logiciel.
- Savoir modifier les paramètres du système si les ressources IPC viennent à manquer pour exécuter les processus applicatifs. L'administrateur qui n'est pas un spécialiste de la programmation système consulte la documentation du logiciel pour déterminer le nombre de ressources IPC qui convient.

Les catégories d'IPC

Les IPC appartiennent à trois catégories distinctes :

- Les files d'attente de message (« *Message Queue* ») que les processus utilisent pour échanger des données. Les files d'attente de message sont des « FIFO ». Elles offrent comme avantages principaux sur les tubes de permettre une gestion de priorité entre messages et de consulter la présence de messages, sans les lire pour autant.

- Les zones de mémoire partagées (« *Shared Memory* ») par plusieurs processus permettent à plusieurs processus de lire ou d'écrire des données dans une zone commune.
- Les tableaux de sémaphore permettent aux processus de se synchroniser pour éviter des conflits d'accès à des ressources partagées, des zones de mémoire partagées par exemple.

Les commandes

Il existe deux commandes qui traitent des IPC :

La commande `ipcs` qui les visualise :

```
# ipcs       # les informations essentielles des trois catégories
# ipcs -q    # les informations essentielles des files de messages
# ipcs -m    # les informations essentielles des mémoires partagées
# ipcs -s    # les informations essentielles des sémaphores
```

Dans tous les cas, la commande `ipcs` affiche, en colonne, des informations relatives aux IPC existantes :

key — La clé désigne le nom de l'IPC. Il a été choisi par le créateur de l'IPC.

id — L'identificateur de l'IPC désigne une valeur attribuée par le noyau à la création de l'IPC. C'est l'équivalent du « handle » d'un fichier. L'en-tête de colonne est respectivement shmid, semid et msqid pour les zones de mémoire partagée, les sémaphores et les files d'attentes de messages.

owner — Le nom du propriétaire de l'IPC.

perms — Le champ perms définit, sous forme numérique, les droits d'accès à l'IPC. La lecture en est identique à celle des droits des fichiers.

bytes | nsems | used_bytes

Selon l'en-tête de colonne, le champ désigne respectivement :

- La taille de la zone de mémoire partagée (bytes).
- Le nombre de sémaphores d'un tableau de sémaphores.
- Le nombre d'octets occupés dans une file d'attente de messages.

Les colonnes supplémentaires des zones de mémoire partagées sont respectivement le nombre de processus attachés à cette zone (nattach) et l'état de la zone (status).

La colonne supplémentaire des sémaphores est l'état de la zone (status).

La colonne supplémentaire des files d'attente de messages est le nombre de messages en attente.

```
# ipcs
------ Shared Memory Segments --------
key              shmid     owner     perms     bytes     nattch
status
0x00000000        0   nobody         600   46084           11
dest
------ Semaphore Arrays --------
key              semid      owner     perms     nsems     status
0x0000000a        0        root          666             1
------ Message Queues --------
key              msqid     owner     perms     used-bytes   messages
0x00000000        0        root          700          0        0
```

La commande ipcrm qui permet de supprimer une ressource IPC.

ipcrm type_IPC ID_de_la_ressource_de_type_x_a_détruire

Le symbole peut être :

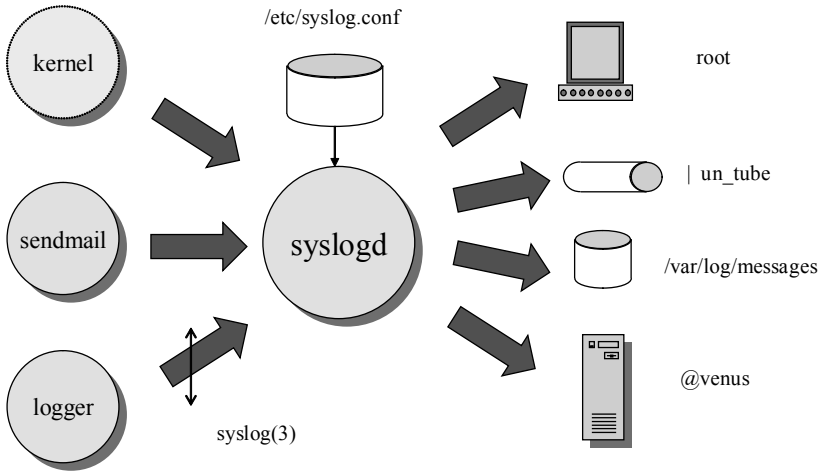
- msg pour les files d'attente de message,
- shm pour les zones de mémoire partagée,
- sem pour les sémaphores.

Références

Man

ipcs(8), ipcrm(8)

Le service Syslog



Introduction

Le démon `syslogd` reçoit des messages d'erreur émis par le noyau ou les démons de certains services, les sous-systèmes. Le fichier */etc/syslog.conf* permet à l'administrateur de décider de la destination des messages pour chaque sous-système et selon le degré de sévérité de l'erreur.

Le fichier syslog.conf

Sur chaque ligne du fichier */etc/syslog.conf* qui n'est pas un commentaire, on indique, comme premier champ, une liste des sous-systèmes avec, pour chacun d'entre eux, le niveau de sévérité du message. Le second champ indique la destination du message.

La structure d'une ligne est donc formellement définie de la manière suivante :

Priorité [; priorité …] action

Un champ priorité est de la forme :

Sous-système[, sous-système …] opérateur niveau

Le caractère * utilisé pour désigner un sous-système signifie tous les sous-systèmes.

Le caractère * comme niveau signifie tous les niveaux.

Le champ opérateur peut avoir les valeurs suivantes :

. Désigne tous les niveaux supérieurs ou égal à, ainsi `mail.err` représente tous les messages du sous système mail supérieur ou égal à err.

. ! C'est la négation de l'opérateur précédent, ainsi `mail.err` représente tous les messages de niveau inférieur à err

.= Désigne tous le niveau identique à, ainsi `mail.=err` désigne tous les messages de niveau err.

.!= Désigne tous les niveaux différents de, ainsi mail!=err représente tous les niveaux différents de err.

Le champ action désigne la destination du message.

La première ligne de l'exemple du transparent montre que les messages de niveau supérieur ou égal à erreur de tous les sous-systèmes et de niveau supérieur ou égal à notification du sous-système d'authentification sont affichés sur la console maîtresse.

La seconde ligne nous dit que ces messages ainsi que les messages de niveau supérieur ou égal à notification du sous-système daemon sont dirigés vers le fichier */var/log/messages*. La troisième ligne nous dit que les messages de niveau supérieur ou égal à debug du sous-système lpr sont envoyés à root. La quatrième ligne nous dit que les messages urgents de tous les sous-systèmes sont envoyés à tous les utilisateurs.

Une configuration minimaliste, consisterait à envoyer tous les messages dans le fichier */var/log/messages*.

```
# more /etc/syslog.conf
*.*      /var/log/messages
```

Dans l'exemple suivant, la première ligne montre que les messages de niveau égale ou supérieur à err (messages d'erreur) de tous les sous-systèmes et les messages de niveau égale ou supérieur à notice du sous-système d'authentification sont affichés sur la console maîtresse. La seconde ligne nous dit que ces messages ainsi que les messages du sous-système daemon (de niveau notice et au-delà), mais pas les messages du sous-système mail, sont dirigés vers le fichier */var/log/messages*.

La troisième ligne nous dit que les messages de type debug du sous-système lpr sont envoyés à root. La quatrième ligne nous dit que les messages urgents de tous les sous-systèmes sont envoyés à tous les utilisateurs. La cinquième ligne nous dit que les messages de niveau égal ou supérieur à err concernant la messagerie sont envoyé dans le fichier /var/log/mail. Les écritures dans ce fichier sont tamponnées, ce qui améliore les performances mais on a le risque de perdre les dernières écritures en cas d'arrêt brutal.

```
# more /etc/syslog.conf
*.err;auth.notice                              /dev/console
*.err;daemon,auth.notice;mail.none             /var/log/messages
lpr.debug                                      root
*.emerg                                        *
mail.err                                       -/var/log/mail
```

Les sous-systèmes

Les sous-systèmes possibles sont :

Sous-système	*Signification*
auth, authpriv	Le service de sécurité et d'authentification. Le terme « auth » est obsolète, il faut maintenant utiliser « authpriv ».
cron	Le service cron.
daemon	Les démons du système.
kern	Le noyau.
lpr	Le service d'impression.
ftp	Le service FTP de transfert de fichiers.

mail	Le service de messagerie.
news	Le sous-système réseau.
syslog	Le démon syslog lui-même.
user	Messages envoyés par les processus des utilisateurs.
uucp	Le sous-système uucp (« *Unix to Unix CoPy* »).
local0 à local7	huit possibilités réservées à un usage local.

Les niveaux de sévérité

Les niveaux de sévérité sont dans un ordre décroissant d'importance :

Niveau	*Signification*
emerg	Le système est inutilisable.
alert	Une intervention immédiate est indispensable.
crit	Erreur critique pour le sous-système.
err	Erreur de fonctionnement.
warning	Avertissement.
notice	Evénement normal mais qui mérite d'être signalé.
info	Pour information seulement.
debug	Message envoyé pour la mise au point.
none	Ignorer les messages.

Remarque

Un niveau de sévérité englobe les niveaux de sévérité supérieurs, ainsi le niveau err englobe les niveaux crit, alert, et emerg.

Les messages sont écrits par le noyau dans /dev/error et par les sous-systèmes dans le tube nommé */dev/syslog*.

Des messages distants sont reçus sur la « socket » liée au port 514.

Les actions

Le champ action peut être :

- Un fichier.
- Un tube nommé, par exemple : | un_tube. Ce tube doit être créé par la commande `mkfifo` avant le démarrage de `syslogd`.
- Un hôte distant, le champ a alors la forme @hôte.
  Le démon `syslogd` de l'hôte distant doit avoir été démarré avec l'option « -r » pour accepter les messages entrants.
- Une liste d'utilisateurs connectés : util[,util…].
- Le caractère * pour désigner tous les utilisateurs connectés.

Envoyer des messages au service Syslog

La commande `logger` ou la fonction syslog() du langage C peuvent être utilisées pour envoyer des messages à `syslogd`.

Il est possible d'associer ce message à une priorité, l'exemple suivant envoie un message associé à la priorité ftp.info :

```
# logger -p ftp.info "Activation du service ftp anonyme"
# tail /var/log/messages
Feb 17 20:42:11 carapuce sendmail: DÃ©marrage de sendmail succeeded
Feb 17 23:05:06 carapuce su(pam_unix)[11682]: session opened for
user pierre by root(uid=0)
Feb 18 00:01:26 carapuce su(pam_unix)[11682]: session closed for
user pierre
Feb 18 00:01:51 carapuce root: Activation du service ftp anonyme
```

Pour aller plus loin, le démon klogd

Le démon `klogd` prend en charge les messages du noyau. Dans sa configuration standard `klogd` envoie les messages au démon `syslogd`.

Le démon `klogd` récupère les messages du noyau grâce au système de fichiers /proc et en particulier au fichier */proc/kmsg*.

```
# ps -e | grep klogd
  494 ?        00:00:00 klogd
```

Parmi les nombreuses options de démarrage du démon `syslogd`, nous avons retenu :

-f fichier	Quand `klogd` est démarré avec cette option, il écrit les messages dans f plutôt que de les transmettre à `syslogd`.
-s	Avec cette option, `klogd` utilise des primitives pour récupérer les messages dans les buffers du noyau plutôt que d'utiliser */proc*. Cette option peut être utile si */proc* n'est pas monté.
-2	Grâce à cette option, les lignes qui traitent des symboles sont écrites deux fois : une première sous forme symbolique et une seconde sous forme brute pour que des outils comme `ksymoops` puissent travailler. La commande `ksymoops` est disponible après installation du paquetage (ksymoops-2.4.9-1.i386.rpm) ou du fichier `tar` compressé. Elle permet d'analyser les messages du noyau après un crash.

Références

Man

syslogd(8), klogd(8), syslog.conf(5), klogd(8), logger(1), syslog(2), syslog(3)

Atelier 7 : La gestion des processus

Objectifs :

- **Savoir lancer et arrêter une application**
- **Savoir gérer le service cron pour activer des tâches périodiques**

Durée : 30 minutes.

Exercice n°1

Est-ce que l'application « crond » est active ? Si oui, quel est son PID ?

Exercice n°2

On exécute la commande suivante :

$ sleep 200 &

a) Quel est son père ?
b) Tuer cette application.

Exercice n°3

L'utilisateur pierre étant connecté, tuez son shell de connexion.

Exercice n°4

Comment autoriser tout le monde à utiliser la commande `crontab` ?

Exercice n°5 (suite de l'exercice précédent)

Comment interdire uniquement à Paul d'utiliser la commande `crontab` ?

Exercice n°6

Quel est le nom du fichier *log* utilisé par le démon `cron` ?
Visualisez-le.

Exercice n°7

Créez, sous le compte de pierre, un fichier *crontab* qui réalise les actions suivantes :

a) Ajoute toutes les minutes du message « Bonjour » suivi de la date, au fichier */tmp/log*.
b) Liste des processus tous les ¼ d'heure, de 8h à 17h du lundi au vendredi, dans le fichier */tmp/processus*.

Listez le *crontab* actif et supprimez-le après avoir constaté qu'il fonctionne.

Exercice n°8

Sauvegardez le fichier *crontab* de l'administrateur, et ensuite ajoutez-y l'action suivante :

Ajout toutes les vingt minutes de la date au fichier */tmp/log*.

Cette action doit être exécutée avec les droits de l'utilisateur pierre.

Exercice n°9

En analysant le résultat de la commande `ps aux`, indiquez, selon votre avis, quelles sont les applications susceptibles d'être des démons.

Exercice n°10

Ajoutez la variable d'environnement « PAYS=France » à tous les utilisateurs.

Exercice n°11

Quelles sont les commandes gérant les IPC sur votre système ?

Exercice n°12

Quelles sont les bibliothèques dynamiques respectivement utilisées par les commandes `cp` et `kedit` ?

Exercice n°13

Utiliser la commande `logger` pour copier le message « serveur xxx a démarré » sur le fichier log de votre système. Lisez ensuite le message généré.