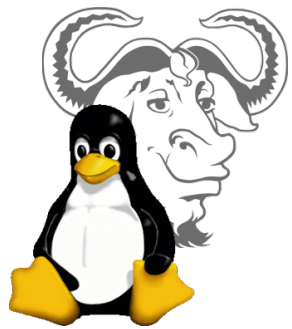
Administration d'un système GNU / Linux

04 — Outils plus avancés

Anthony Labarre

上海师范大学

10 novembre 2022



Outils bons à connaître

Nous allons présenter brièvement quelques outils qui vous seront très utiles sur les thèmes suivants :

1. Programmation de tâches
2. Sauvegardes
3. Programmation et comparaison
4. Commandes "*offline*"
5. Épilogue

Il y aura peu de détails : le but est de vous mettre au courant de l'existence de ces outils.

`cron`

- Il est souvent utile de programmer l'exécution de certaines tâches ;
- Par exemple : "sauvegarder mes fichiers tous les jours à 17h30" ;
- Il existe principalement deux outils standards pour ceci :
  1. `cron`, pour une programmation précise ;
  2. `anacron`, pour une programmation différée ;

`cron`

- `cron` est un démon qui permet de planifier l'exécution de tâches à des moments précis ;
- On précise pour chaque entrée du fichier de configuration :
  1. l'heure de début ;
  2. les jours ou mois d'exécution (par défaut : tous les jours) ;
  3. l'utilisateur qui lancera la tâche ;
  4. et la commande à exécuter ;

Configuration globale ou locale de `cron`

- Des tâches peuvent être programmées de deux façons (on peut combiner les deux) :
  1. **globale :** pour tous les utilisateurs ; ou
  2. **locale :** chaque utilisateur peut aussi planifier des tâches juste pour lui ;
- La configuration globale est dans `/etc/crontab` ;
- Les tâches locales se configurent avec la commande `crontab -e` ;

Exemple de configuration dans /etc/crontab

Le format d'une entrée est structuré comme suit :

[moment d'exécution] [utilisateur] [commande à exécuter]

Exemple (fichier /etc/crontab)

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

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 * * * * root    cd / && run-parts --report /etc/cron.hourly
25 6 * * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6 * * 7 root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6 1 * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
```

Avantages et inconvénients de `cron`

- ✓ `cron` exécute les commandes planifiées ;
- ✓ On peut les paramétrer avec une grande précision (jusqu'à la minute) ;
- ✓ `cron` est disponible sur tous les systèmes GNU / Linux ; mais ...
- × Si votre machine est éteinte au moment où la tâche doit s'exécuter, elle ne s'exécutera pas !

anacron

- `anacron` permet également de paramétrer l'exécution de tâches périodiques ;
- La différence avec `cron` :
  - `cron` fixe un moment précis (tous les jours à 17h30) ;
  - `anacron` fixe une période (tous les jours, "quand c'est possible") ;
- `anacron` est donc préférable si :
  - votre machine n'est pas allumée en permanence ;
  - l'heure précise d'exécution n'est pas importante ;

Configuration globale ou locale de anacron

- Comme pour cron, des tâches peuvent être programmées de façon **globale** ou **locale** ;
- La configuration globale est dans /etc/anacrontab ;
- Pour les tâches locales, on "imite" la configuration globale :

```
$ mkdir -p ~/.anacron/{etc,spool}
$ echo $SHELL >> ~/.anacron/etc/anacrontab
$ echo $PATH >> ~/.anacron/etc/anacrontab
```

- Puis on utilise crontab -e pour exécuter anacron toutes les heures (par exemple) avec l'entrée :

```
@hourly /usr/sbin/anacron -s -t $HOME/.anacron/etc/anacrontab \
    -S $HOME/.anacron/spool
```

Exemple de configuration dans `/etc/anacrontab`

Le format d'une entrée est structuré comme suit :

[période] [délai] [job-ID] [commande]

Exemple

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
HOME=/root
LOGNAME=root

# These replace cron's entries
1       5       cron.daily      run-parts --report /etc/cron.daily
7       10      cron.weekly     run-parts --report /etc/cron.weekly
@monthly        15      cron.monthly    run-parts --report /etc/cron.monthly
```

- `période` : numéro(s) du (des) jour(s) ou `@daily`, `@weekly`, and `@monthly` ;
- `délai` : nombre de minutes à attendre avant de démarrer ;
- `job-ID` : l'identifiant unique de la tâche prévue ;

Utilité des sauvegardes

- Il est capital de réaliser des sauvegardes fréquentes des fichiers importants ;
- De nombreux utilisateurs croient que les sauvegardes sont inutiles . . .
- . . . jusqu'au jour où un gros problème leur fait perdre toutes leurs données (disque dur défectueux, effaçage accidentel, . . .)
- Quel(s) outil(s) utiliser ?

Outils de sauvegarde

- Il existe de nombreux programmes réalisant des sauvegardes, avec différents “niveaux d’intelligence” :
  - tout recopier à chaque sauvegarde ;
  - ou recopier uniquement les fichiers nouveaux ou modifiés ;
  - ou recopier uniquement les **changements** ;
- Un des plus recommandés : `rsync` ;

`rsync`

- `rsync` est un outil de sauvegarde possédant de nombreux avantages, notamment :
  - la sauvegarde / restauration locale ou à l'aide d'un serveur ;
  - l'utilisation des *différences* entre exemplaires de fichiers pour la sauvegarde ;
  - extrêmement configurable ;
- De nombreux outils lui servent d'interface et sont plus faciles à configurer, notamment `backintime` ;

`comm`

- `comm` est un outil de comparaison de textes **triés**
- sans option, la commande `comm premier second` affiche trois colonnes côte à côte :
  1. les lignes n'apparaissant que dans `premier` ;
  2. les lignes n'apparaissant que dans `second` ;
  3. les lignes apparaissant dans `premier` et `second`.

comm

- En option, on peut préciser les numéros des colonnes à supprimer ;
  - `premier` $\setminus$ `second` = `comm -23 premier second` ;
  - `second` $\setminus$ `premier` = `comm -13 premier second` ;
  - `premier` $\cap$ `second` = `comm -12 premier second`.

`diff`

- `diff premier second` affiche les différences entre deux fichiers ;

Exemple

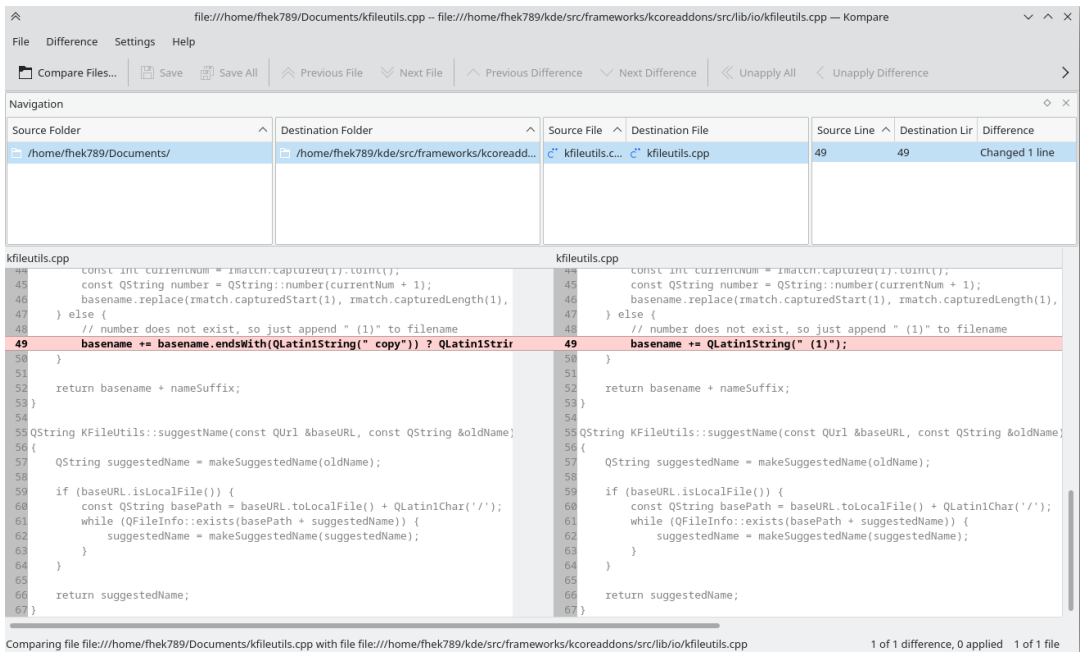
```
$ sed s/Quasimodo/Superman/g notredamedeparis.txt > superman-de-paris.txt
$ diff notredamedeparis.txt superman-de-paris.txt
2138,2139c2138,2139
< ``C'est Quasimodo, le sonneur de cloches! c'est Quasimodo, le bossu de
< Notre-Dame! Quasimodo le borgne! Quasimodo le bancal! Noël! Noël!''
---
> ``C'est Superman, le sonneur de cloches! c'est Superman, le bossu de
> Notre-Dame! Superman le borgne! Superman le bancal! Noël! Noël!''
```

- `diff` indique les lignes et les caractères dans les deux fichiers où des différences apparaissent ;
- < marque les lignes de `premier`, > marque les lignes de `second` ;

`diff`

- Des *frontends* pour `diff` permettent de visualiser plus facilement les différences ;

Exemple (source : `https://apps.kde.org/kompare/`)



- `diff` nous servira aussi à produire des *patches* (voir suite) ;

Patches

- Un ***patch*** est une modification que l'on rajoute à un logiciel, généralement pour corriger un bug, une erreur ou un oubli ;
- Ils sont très fréquents en informatique :
  - les mises à jour de sécurité (Windows ou autres) ;
  - les corrections dans les jeux vidéo ou l'ajout de contenu ;
  - ...
- En général, ils sont distribués sous forme binaire ; mais avec les logiciels libres, on peut aussi distribuer les corrections des sources !
- Intérêt : les *patches* sont beaucoup plus petits que la nouvelle version entière du logiciel ;

`diff`

- `diff` ne sert pas uniquement à détecter la fraude chez les étudiants !
- Souvent, on corrige des bugs dans les programmes, et `diff` permet de repérer ce qui a changé ;
- On stocke les différences dans un fichier ; par exemple : `diff -u premier second > changements.patch`
- On peut maintenant distribuer uniquement `changements.patch`, que les utilisateurs pourront utiliser pour corriger leur version de `premier` ;

`patch`

- `patch` permet d'appliquer les changements repérés avec `diff` ;
- Si l'on possède le premier fichier, on n'a pas besoin du second, juste de `changements.patch` ;
- On les applique avec la commande
  `patch premier changements.patch`
- Il est possible aussi d'annuler les changements (voir `man patch`) ;

make

- Les commandes que l'on écrit deviennent de plus en plus compliquées ;
- Exemple : pour compiler les slides que vous êtes en train de lire :

```
latexmk -f -jobname=$(basename $<)-handout -pdf \
    -pdflatex='pdflatex -jobname=$(basename $<)-handout
    -shell-escape -interaction=nonstopmode
    "\PassOptionsToClass{handout}{beamer}
    \input{$(basename $<)}"' $<
```

- . . . personne n'a envie de taper ces longues commandes et encore moins de les retenir ;

Solution : `make`

- `make` permet de paramétrer les commandes qui nous intéressent et les enregistrer dans un fichier nommé `Makefile` ;
- Avantages :
  - pour compiler les sources, il suffit de taper `make`
  - `make` ne recompile que les parties nécessaires ;
  - tous les langages de programmation sont utilisables (C, . . .) ;

Format d'un `Makefile`

- Les `Makefile` peuvent être très compliqués, mais le principe est toujours le même ;
- On définit des **cibles** (= *targets*), qui sont les résultats à obtenir ;
- Pour chaque cible, on définit des **règles** (= *rules*) de construction, qui expliquent à `make` comment construire les cibles ;

Les cibles

Les cibles doivent être écrites de la manière suivante :

```
nom_de_la_cible : liste de prérequis
	règle 1
	règle 2
	...
```

- `nom_de_la_cible` est un mot sans espaces
- `liste de prérequis` est une liste d'autres cibles qui doivent avoir été construites ;
- `règle 1`, `règle 2`, ... sont les instructions de compilation de la cible ;

Attention, les tabulations sont importantes ! **Ne les remplacez pas par des espaces !**

Exemple de `Makefile`

```
edit : main.o kbd.o command.o display.o \
       →insert.o search.o files.o utils.o
       →cc -o edit main.o kbd.o command.o display.o \
       →       →insert.o search.o files.o utils.o

main.o : main.c defs.h
       →cc -c main.c
kbd.o : kbd.c defs.h command.h
       →cc -c kbd.c
command.o : command.c defs.h command.h
       →cc -c command.c
display.o : display.c defs.h buffer.h
       →cc -c display.c
insert.o : insert.c defs.h buffer.h
       →cc -c insert.c
search.o : search.c defs.h buffer.h
       →cc -c search.c
files.o : files.c defs.h buffer.h command.h
       →cc -c files.c
utils.o : utils.c defs.h
       →cc -c utils.c
clean :
       →rm edit main.o kbd.o command.o display.o \
       →       →insert.o search.o files.o utils.o
```

(source : https://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html)

Tâches longues

- On a souvent besoin d'accéder à des machines beaucoup plus puissantes que la nôtre pour effectuer des calculs ;
- En général, on se connecte en `ssh`, puis on lance la tâche dans le terminal ;
- Problème : quand on se déconnecte, la tâche se termine !
- Comment faire si le programme doit tourner plus longtemps (jours, semaines, mois, . . .) ?

nohup

- Solution : lancer la tâche avec `nohup` ;
- `nohup commande arguments` lance la commande comme d'habitude ;
- Mais la tâche continue à s'exécuter même si vous vous déconnectez ;
- En général, on redirige la sortie vers un fichier puisqu'on ne prévoit pas d'attendre l'affichage du résultat :
  `nohup commande arguments > fichier`
- On peut aussi recevoir un e-mail quand la tâche est finie ;

Quoi d’autre ?

Le cours s’arrêtera là, mais il reste énormément de choses à explorer :

- “forensics” : la récupération de données “effacées” ou endommagées ;
- la récupération du système en cas de crash grave (*kernel panics*, disques introuvables, . . .)
- compiler son propre noyau sur mesure ;
- l’administration réseau ;
- sécuriser son système ;
- . . .