Automating administrative tasks

Grado en Informática 2023/2024

Departamento de Ciencias de la Computación
y Tecnologás de la Información
Facultad de Informática
Universidad de Coruña

Antonio Yáñez Izquierdo

Contenidos I

Contenidos II

- introduction: logs, logfiles and syslogd
- log configuration
- rotating of logs
  - solaris logadm
  - BSD newsyslog
  - linux logrotate

5 Starting and stopping system services. boot scripts
- system V init scripts
- linux
  - devuan
  - fedora linux
  - debian
  - ubuntu
- solaris
- openbsd

Contenidos III

- freebsd
- netbsd

Automating administrative tasks

Automating

- Many of the tasks of the system administrator must be carried out more than once
  - backups
  - checking logs for errors
  - checking free space in partitions
  - ...
- Fortunately most of them can be automated. There are two sides to this automation
  - writing *scripts* to perform the tasks
  - arranging for the *scripts* execution to be automated
  - checking the execution of tasks using the *logs*

Shell scripting

introduction
basic scripting
operations
conditional execution
loops
text manipulation
sed

Scripts and shells

- By script we usually refer to a plain text file with commands understandable by some interpreter: perl, python ...
- Unix shells (bash, csh, ksh, sh, tcsh ...) are, in fact, interpreters
- A shell script consists of some shell commands written in a text file to perform repetitive tasks that we usually could do directly from a terminal.
- They can also be used to process text files

Scripts and shells

- They are widely used in Unix systems: initialization scripts, installation scripts ...
- We can classify shells in two groups, according to scripting syntax
  - **sh-like shells**: `sh`, `ksh`, `bash`, `ash`, `dash`
  - **csh-like shells**: `csh`, `tcsh`
- As it is more commonly used, we'll deal here with *sh-like* scripting

Structure of a shell script

- Here we have a sample shell script

```
#!/bin/sh

command1
command2
command3 ; command4
....
# this is a comment
```

- The first line, after the symbols '**#!**', indicates which program is interpreting the script

Structure of a shell script

- We use one line for each command
  - we can also put more than one command in one line, separated by ';'
- Comments start with '#', an go to the end of the line
- The script must have execution permission

Shell scripts: output and input

- We can use the command **echo** to produce output:

  ```
  echo [-n] text
  ```

- The *-n* option does not write and *end of line*
- We can use '**read**' to get input from the keyboard. Example

  ```
  echo -n "Enter directory name: "
  read DIR
  ```

Shell scripts: variables

- Variables:
  - They need not be declared and are typeless
  - A variable is identified by its name (Example: *foo*), but we must use the symbol '**$**'. to access its value (Example: *$foo*).
  - Uppercase and lowercase letters are significant in variable names
  - Some variables are predefined (environment variables). For example *$HOME*, which provides the user's home directory
  - To assign a value to a variable we use the symbol '**=**' without spaces before or after it. Example

    ```
    DIRECTORY=/usr/local/bin
    ```

Quotation marks

- Text can be surrounded by three different quotation marks, with very different meanings:
  - **simple quotes: 'Text'** Text is interpreted literally. No variables are substituted and no characters are interpreted.
  - **double quotes:"Text"** Variables are substituted and characters are interpreted.
  - **inverted quotes: ‘Text‘** Variables are substituted and characters are interpreted, *Text* is interpreted as a command and executed. The value of ‘Text‘ is the output of that command

```
FILE=/home/user/data
PRU='wc -l $FILE'#wc -l $FILE
PRU="wc -l $FILE" #wc -l /home/user/data'
PRU=‘wc -l $FILE‘#457
```

Command line arguments and special variables

- Command line arguments: they can be accessed inside a shell script as variables $1...$9.
- If there are more than 9 command line arguments we can use *shift*
- '**$#** has the number of command line arguments the shell script has received
- '**$$**' pid of the shell executing the script
- '**$?**' return value of last command executed
- '**$IFS**' List of Internal Field Separator characters

Redirecting input, output and error

- Standard input and output can be redirected to a file
  - $>$ redirects standard output to a file.
  - $<$ redirects standard input to a file.
  - $>>$redirects standard output to a file, appending the standard output to the existing contents of the file.

Redirecting input, output and error

- $>$ & or 2 $>$ Redirects the standard error to a file (depends on the shell)
- The standard output of one process can be redirected to the standard input of another process, thus communicating processes with the symbol '|'. Examples

```
$ ls -l | more
$ cat *.c | wc -l
```

Operating with **expr**

- To perform operations we can use the command **expr**.
- **expr EXPRESSION** prints to the standard output the result of EXPRESSION; EXPRESSION can be
  - `arg1 | arg2` arg1 if arg 1 not null or zero, otherwise arg2.
  - `arg1 & arg2` arg1 if arg1 is null or 0, otherwise arg2.
  - `arg1 < arg2` 1 if arg1 < arg2, otherwise 0. Relational operators (`<`,`<=`,`>`,`>=`,`=`,`!=`) operate both on integers and on strings.
  - `arg1 + arg2`. Aritmethic sum. (Other arithmetic operators:`-`,`*`,`/`,`%`).
- Example:

```
#!/bin/sh
x=‘expr 4 + 5‘
echo $x
```

Operating with **let**

- In some shells (for example bash) we can use '**let**' to calculate the result of operations
- Available operations are
  - Basic arithmetic: +, -, *, /, %
  - Increment: id++, –id
  - Decrement: id–, –id
  - Exponential: **
  - Displacements: <<, >>
  - Comparison: <, >, <=, >=
  - Equal/not equal: ==, !=
  - Logic operators: &&, ||
  - Bit operators: &, ^ , |
- Example:

```
#!/bin/sh
let x=4+5*3
echo $x
```

Conditional execution: **if**

- **if**
  - Allows conditional execution depending on one condition. The syntax is

```
if condition; then command1; command2; ...
else command4; command5
fi
```

```
if condition1 then ....;
elif condition2 then .... ;
else .... fi
```

Conditional execution: **case**

- **case**
  - Allows for executing different parts of code depending on the value of a variable

```
case word in
pattern1) commands ;;
pattern2) commands ;;
....
esac
```

  - case compares the word with the patterns and executes the code corresponding to the FIRST pattern that matches. Pattern order is thus significant

Conditional execution: **case**

- Example of the use of **case**

```
case $# in
0) ..........  ;;    #no arguments
1) ..........  ;;    #one argument
*) ..........  ;;    #otherwise...
esac

case $1
start) ......  ;;
stop) .....   ;;
*)
  echo use {start|stop}
  exit 0
  ;;
esac
```

Shell scripting: **test**

- **test** or **[..]**: It is used to test if certain conditions are met. The syntax is `test EXPRESSION` or `[ EXPRESION ]` (there are spaces between EXPRESSION and the square brackets)
- EXPRESSION can be
  - '(EXP)' EXP is true
  - '! EXP' EXP es false
  - 'EXP1 -a EXP2' both EXP1 and EXP2 are true
  - 'EXP1 -o EXP2' either EXP1 or EXP2 is true
- EXP can involve numbers, strings or files

Shell scripting: **test**

- Some of the conditions **test** can perform on integers:
  - A -eq B : equal
  - A -ne B : not equal
  - A -gt B : greater than
  - A -lt B : less than
  - A -ge B : greater or equal than
  - A -lt B : less or equal than

Shell scripting:**test**

- Some of the conditions **test** can perform on strings:
  - `S = T` : equal
  - `S != T` : not equal
  - `-n S` : checks if S exists and is not null
  - `-z S` : checks if length of S is 0

Shell scripting:**test**

- Some of the conditions **test** can perform on files
  - f1 -ef f2 : f1 and f2 have the same device and inode
  - f1 -nt f2 : f1 is newer than f2 (-ot for older)
  - -s file : file is of size greater than 0
  - -e file : file exists
  - -f file : file is a regular file
  - -d file : file is a directory
  - -b file : file is a block device
  - -c file : file is a character device
  - -h file : file is a symbolic link
  - -r file : file is readable.
  - -w file : file is writable
  - -x file : file is executable (has execution permission)
  - -u file : file is a setuid file
  - -g file : file is a setgid file

Shell scripting: **for** and **while**

- loops can be done in shell scripts with **for** and **while**
  - **for** allows repeating a task a determined number of times

    ```
    for var in list; do command1;
    command2;.. done
    ```

  - If we want to use a integer list of values, it can be generated with the **seq** command. Example

    ```
    for i in ‘seq 1 100‘; do echo $i; done
    ```

  - **while** allows repeating a task until a condition is met

    ```
    while condition do command1;
    commando2; done
    ```

Shell scripting: text manipulation

- Any external program can be called from a shell script. The most common ones called for text processing are `cut`, `tr`, `sed`, `grep`, `paste`, `cat`, `echo` .... They operate on the standard input and write to the standard output
  - **cut** allows to get certain parts o a text file by getting certain parts of each line.

    `cut -f field -d delimiter. Example: cut -f1,4 -d:`
  - **tr** substitutes or eliminates characters

    `tr [CAR1] [CAR2].`

    Example: `tr a-z A-Z < file` changes lowercase to uppercase in file
    Example: `tr -s " "` deletes repeated occurrences of the space character

Shell scripting: **grep**

- **grep** allows to get certain parts o a text file by getting certain lines
- **grep** selects the lines on the file matching a pattern (regular expression)
- **grep** operates on its standard input (or in a file supplied as command line argument) and writes to its standard output

```
cat file.txt | grep "[0-9]\+\.[0-9]*"
```

Shell Scripting: **sed**

- **sed** is a non-interactive line editor
  - It reads from the standard input or from a file supplied as command line argument
  - Performs a series of operations on some of the lines of the file
  - writes to the standard output
- It determines on which lines it must operate by the range or pattern it is supplied. If no range or pattern is upplied, it operates on all the lines.
- **sed** prints the lines as it is processing them. If we want **sed** to not print the line it is processing, we must use **sed -n**

Shell scripting: **sed**

- We will only comment a few of sed operations
  - print lines: **[range]/p** (to print only the selected lines **sed -n**)
  - delete lines: **[range]/d**
  - substitute strings: **s/pattern1/pattern2/**
  - substitute characters: **y/characters1/characters2/**
- substitution commands only change the first occurrence, if we want to make them global we must use '**g**'. Example **sed s/one/two/g < testfile** substitutes *one* for *two* in file *testfile*

Shell scripting: **sed**

- Line selection in **sed**
  - /regexp/: lines that match regular expresion *regexp*
  - \cregexpc:clines that match regular expresion *regexp* líneas que contienen la expresión regular *regexp* preceeded and followd by character '**c**'
  - n,~M: every **M** lines counting from line **n**
  - li,+M: line **li** and the following **M** lines. **li** can be a line number or a regular expression
  - li,~M: line **li** and the following lines up to an **M** multiple

Shell scripting:**sed** examples

- print lines **[range]p**

```
$sed -n 3,+4p <file
  #prints lines 3,4,5,6,7
$sed -n /que/p
  #prints lines containing 'que'
$sed -n /hello/,~3p < file
  # from the first line containing 'hello'
  # to the first 3 multiple
```

Shell scripting:**sed** examples

- delete lines **[range]d**

```
$sed 3,+4d <file
  #deletes lines 3,4,5,6,7
$sed /que/d
  #deletes lines containing 'que'
$sed /hello/,~3d < file
  # from the first line containing 'hello'
  # to the first 3 multiple
```

Shell scripting:**sed** examples

- substitute strings **s/pattern1/pattern2/**

```
$sed s/hello/HOLA/ < file
```

- substitute characters **y/caracteres1/caracteres2/**

```
$sed y/afk/JKL/ < file
```

other utilities

- there are also many other programs that operate on text files. Some of them
- **head** selects the first part of a file (lines, chars. . .)

```
$ head -15 < fil # the first 15 lines of file fil
```

- **tail** selects the last part of a file (lines, chars. . .)

```
$ tail -20 < fil # the last 20 lines of file fil
```

- **wc** counts lines, words, characters of a file

Shell scripting: functions

- Modern shells allow their scripts to have functions. The syntax to define a function is

```
function_name (){
executable sentences
...
}
```

- a function can receive arguments. Inside a function, arguments are referred by the variables $1, $2 ...
- to invoke a function with arguments we call it as we would call a program with arguments from the command line
- to return a value a function can **echo** the value so it can be assigned to a variable with inverted quoting marks ‘ or the $() construct

Shell scripting: functions example

```
#!/bin/bash
suma (){

  let result=$1+$2
  echo $result

}


VAR1=‘suma 2 2‘
VAR2=$(suma 2 3)

echo VAR1=$VAR1’ ’ VAR2=$VAR2
```

Schedulling execution of tasks: the cron and at commands

crontab files
the crontab command
the at command

the cron service

- after we've made some *fabulous* scripts to perform our tasks we would like them to run autonomously
  - we need not be logged in the machine to invoke the scripts
  - we won't forget to run the scripts
- the **cron** service takes care of this allowing us to
  - arrange for scripts (or programs) to be run periodically
  - arrange for scripts (or programs) to be run once at a certain time

crontab file

- each user in the system has a crontab file that specifies the tasks he/she wants to be executed periodically
- the location of this files varies from system to system
- some systems have a global system crontab file (typically `/etc/crontab`)
- the cron service reads the files when it starts so the users **must not** edit these files directly.
  - they must be edited with the command **crontab -e** which also notifies **cron** that the files have changed

format of the crontab file

- a crontab file consists of a series of lines; one line for each task
- the format of one line is

```
minute   hour    day_of_month   month   weekday   command argumen
```

- in a global crontab file, each line has the format

```
minute hour day_of_month  month weekday user  command arguments
```

- lines starting with # are treated as comments
- some versions allow for environment variables to be set in the crontab file

format of the crontab file

- each of the time fields can have the following characters
  - an integer (0 to 59 for minutes, 0 to 23 for hours, 1 to 31 for day of month, 1 to 12 for month, 0 to 7 for weekday)
  - an *, matching any value
  - several integers separated by ',', thus specifying a discrete set of values
  - two integers separated by '-' to specify a range
  - some systems allow to specify a step with the character '/'

format of the crontab file: example

- the following line executes `/sbin/backup.sh` at 6:30 on days 1, 10, 20 each month, provided they are not a Saturday or Sunday

```
30   6   1,10,20   *   1-5     /sbin/backup.sh
```

- on some systems, the following line executes /usr/bin/check-for-updates every two hours on Sundays and Wednesdays

```
0   */2   *   *   0,3     /usr/bin/check-for-updates
```

- on some systems, the following line executes /home/trump/do-and-say-something-stupid for user *trump* every minute from 9 to 21 every day

```
*   9-21   *   *   *   trump   /home/trump/do-and-say-something-stupid
```

location of the crontab files

- if there exists a global crontab file it is located in `/etc/crontab`
- the crontab file for a user is named after his/her login and its location can be usually found in the online manual page for the **crontab** command
- this location is

linux	`/var/spool/cron/crontabs`, `/var/spool/cron`
solaris	`/var/spool/cron/crontabs`
openBSD	`/var/cron/tabs`
freeBSD	`/var/cron/tabs`
netBSD	`/var/cron/tabs`

the crontab command

- the **crontab** command allows users to modify their crontab. users **must not** edit these files directly
  - **crontab -e** invokes a text editor to edit the user's crontab file. This editor can be specified with the EDITOR environment variable
  - **crontab -l** lists a user crontab
  - **crontab -r** removes a user crontab

cron.allow and cron.deny files

- access of users to the cron facility can be controlled with the `cron.allow` and `cron.deny` files
- each of this files has logins of users, one login per line
- the `cron.allow` file takes precedence over the `cron.deny` file
  - if it exists, users listed in this file are allowed to use the crontab command, the rest of the users are not
- the `cron.deny` is checked if the file `cron.allow` doesn't exist
  - if it exists, users listed in this file are not allowed to use the crontab command, the rest of the users are
- In most of the systems, if neither file exists, all users are allowed to use the crontab command

location of the cron.allow and cron.deny files

- as usual, the location of these files can be usually found in the online manual page for the **crontab** command

linux	`/etc/cron.allow` and `/etc/cron.deny`
solaris	`/etc/cron.d/cron.allow` and `/etc/cron.d/cron.deny`
openBSD	`/var/cron/cron.allow` and `/var/cron/cron.deny`
freeBSD	`/var/cron/cron.allow` and `/var/cron/cron.deny`
netBSD	`/var/cron/cron.allow` and `/var/cron/cron.deny`

other considerations

- on some systems, programs to be run via the **crontab** command must have their output redirected in order to prevent it from being lost.
- however, on other systems, **cron** would mail the user the output of the program
- some systems provide the `/etc/cron.d` directory for software packages to install their crontab files there

other considerations

- linux provides the `/etc/cron.hourly`, `/etc/cron.dayly`, `/etc/cron.weekly` and `/etc/cron.monthly` directories so that scripts can be placed there to be executed periodically
- on solaris, if neither file `cron.allow` nor `cron.deny` exists, the *solaris.jobs.user* authoritation is checked
- on solaris, cron manages its log directly. On linux and BSD it relies on *syslog* for that task
- freeBSD has the *periodic* utility used to run scrips at directories periodically. With this utility, the scripts at `/etc/periodic/daily`, `/etc/periodic/weekly ...` are treated similar to the `/etc/cron.dayly`, `/etc/cron.weekly` directories in linux

the **at** command

- the **at** (or *batch*) command allows a user to submit a job for execution at a later time
- the jobs are scheduled for execution **once**
- **cron** uses the files `at.allow` and `at.deny` to handle the authoritation to submit jobs, in a similar way as it does with the `/etc/cron.allow` and `/etc/cron.deny` files for the **crontab** command

Monitoring system: logs

introduction: logs, logfiles and syslogd
log configuration
rotating of logs

logs

- a *log* is a description of an event that happened to a process in the system
- although some programs can use and maintain their partigular log files it is usual the log daemon in the system (typically named *syslogd*) takes cares of the logs in a centralized way. (linux usually replaces syslog with another *"more advanced"* utility like *syslog-ng* or *rsyslog*)
- usually a *log* is a single line of text containing
  - time and date of the event
  - the machine and service where it has originated,
  - the type and severity of event

logfiles

- a *logfile* is a file where the system stores the logs
- typically is a plain text file containing one line per event
- there can exist different files for different services
- instead of logging to files, logs can also be sent to some device (for example a *terminal*), to users on the system or even to other systems on the network

location of logs files

- the location of the log files varies from system to system. Nearly every system has them under the `/var` directory
- the location of the files can also be defined by the system administrator. The following lines list the most often used default locations on different systems
- **linux**: they are stored in **/var/log/***
- **solaris**: two generic locations: **/var/adm/log/*** **/var/log/***. And plenty of particular locations **/var/cron/log**, **/var/saf/***, **/var/lp/logs/***, **/var/svc/log**
- ***BSD**: most of them are stored under **/var/log/***

syslogd

- *syslogd* is the daemon that takes care of the logs on the system
- applications submit messages to *syslogd*
- *syslogd* reads its configuration file when it starts and decides what to do with the messages it receives
- there are alternatives to syslogd, most of them used on linux systems: *rsyslog*, *dsyslog*, *syslog-ng* . . .

log configuration

- for syslogd (or any of its alternatives) to know what to do with the messages, it must be specified in its configuration file.
- this file is typically **/etc/syslog.conf** (**/etc/rsyslog.conf** if rsyslog is being used . . .)
- a log message is classified according to
  - its *facility*: which service has generated the log. One of a predefined list on the system.
  - its *severity*: how important the log is. One of a predefined list on the system.

syslog facilities

- this are the more usual facilities on syslog

auth	security/authorization messages
authpriv	security/authorization messages (private)
cron	cron and at
daemon	system daemons without separate facility value
ftp	ftp daemon
kernel	kernel messages
lpr	line printer subsystem
mail	mail subsystem
news	USENET news subsystem
syslog	messages generated internally by syslogd(8)
user	generic user-level messages
uucp	uucp subsystem (obsolete)

syslog severities

- this are the more usual severities on syslog

emerg	system is unusable
alert	action must be taken immediately
crit	critical conditions
err	error conditions
warning	warning conditions
notice	normal, but significant, condition
info	informational message
debug	debug-level message

syslog file format

- each line of the file specifies what to do with some logs. Lines starting with # are treated as comments
- the format of one lines is

  `selector  <tab>  action`

- selector selects logs based on the facility and severity. It has the form `facility.severity`.
  - some systems accept the * as a wildcard for facility and/or severity
  - some systems also accept the format `facility1,facility2.severity` or `facility1.severity1; facility2.severity2`

syslog file format

- action represents what must be done with the log selected by *'selector'*. It can be one of the following
  - write the log to a file. This is represented by the name of the file (starting with /). A log can also get sent to a device (for example a terminal) using the device name as the logfile
  - notify users. In this case, action is a comma separated list of users that would get the log provided they are logged in the system. Usually the symbol * stands for all users

syslog file format

- send the log to another machine running *syslogd*. If action starts with **@** the log is sent to the machine specified after the character **@** (name or ip). logs coming from other machine do not get resent to another
- syslog uses port 514 over UDP, some syslog replacements allow the sending of logs using TCP. On those systems we typically would put in the configuration file **@@**`name-or-ip` of the machine receiving the log

```
cron.emrg;cron.alert @192.168.1.5
cron.alert root,cronmaster
cron.err /var/log/cron-errors.log
cron.* /var/log/cron.log
##log over TCP, not available in every system
cron.* @@192.168.1.22
```

extensions

- there are a number of functionalities that, although not standard, can be found on many systems, (specially on linux systems, where a great number of *syslogd* alternatives are available)
  - the existence a directory (typically `/etc/syslog.d`) where different software packages can place their particular log configuration
  - the possibility of, instead writing the logs to a file (or sending them to another machine), start a program and pass the log to its standard input
  - use of conditional sentences as actions. For example `ifdef('LOGHOST', /var/log/syslog, @loghost)`

rotating of logs

- the problem with log files is that they keep growing in time. Large files use up a lot of disk space and are difficult to manage
- the solution is to *rotate* the logs: create a new file once the log file has a certain size or a certain age.
- there are several *log rotating* programs

solaris **logadm** is the log rotating program supplied with *solaris* although *logrotate* can also be installed

bsd **newsyslog** is the default rotating program that comes with openBSD, netBSD and freeBSD

linux **logrotate** is the standard log rotating program

solaris logadm

- *logadm* is the program for rotating the logs in solaris
  a. it can be invoked without arguments, in which case it reads its configuration file, `/etc/logadm.conf`
  b. it can be invoked with a log file as argument, in which case it rotates that log file (adding suffixes 0, 1 . . .)
- the most common use is *a)*, where it is started through *crontab* (typically once a day) and rotates the files specified in its configuration file
- when invoked from the command line to rotate a single file, its syntax is

```
logadm options logfile
```

solaris logadm

- the most common options when invoked with arguments are
  - **-a command** execute command after rotating the log file
  - **-b command** execute command before rotating the log file
  - **-A age** delete versions of the file that have not been modified for the amount of time represented by *age*
  - **-c** instead of renaming the log file to rotate, copy it and truncate the original to zero length
  - **-C count** keep count files
  - **-g grp** use *grp* as the new logfile group
  - **-m mod** use *mod* as the new logfile mode
  - **-o owner** use *owner* as the new logfile uid
  - **-p period** rotate the file after *period*
  - **-s sz** rotate the file if its size is greater than sz

solaris logadm configuration file

- *logadm* si typically invoked without arguments through *crontab*
- in this case it reads its configuration file, `/etc/logadm.conf` unless it is invoked with the `-f config_file` option
- the format of *logadm* config file is
  - lines starting with # are treated as comments
  - lines are in the format

    ```
    logfile  options
    ```

  - the options are expressed as they would be if *logadm* was invoked from the command line

sample solaris logadm configuration file

```
# The format of lines in this file is:
#       <logname> <options>
# For each logname listed here, the default options to logadm
# are given.  Options given on the logadm command line override
# the defaults contained in this file.
#
# logadm typically runs early every morning via an entry in
# root's crontab (see crontab(1)).
#
/var/log/syslog -C 8 -a 'kill -HUP `cat /var/run/syslog.pid`'
/var/adm/messages -C 4 -a 'kill -HUP `cat /var/run/syslog.pid`'
/var/cron/log -c -s 512k -t /var/cron/olog
/var/lp/logs/lpsched -C 2 -N -t '$file.$N'
/var/fm/fmd/errlog -N -s 2m -M '/usr/sbin/fmadm -q rotate errlog && mv /var/fm/fmd/errlog.0- $nfile'
/var/fm/fmd/fltlog -N -A 6m -s 10m -M '/usr/sbin/fmadm -q rotate fltlog && mv /var/fm/fmd/fltlog.0- $nfile
smf_logs /var/svc/log/*.log -C 8 -s 1m -c
#
# The entry below is used by turnacct(1M)
#
/var/adm/pacct -C 0 -N -a '/usr/lib/acct/accton pacct' -g adm -m 664 -o adm -p never
#
# The entry below manages the Dynamic Resource Pools daemon (poold(1M)) logfile.
#
/var/log/pool/poold -N -s 512k -a 'pkill -HUP poold; true'
```

BSD newsyslog

- *newsyslog* is used in openBSD and freeBSD to rotate the logfiles when they reach a certain size or age
- rotated files are renamed .0, .1 ...
- it is usually run by *cron* although it can be also be run manually
- unless it is invoked with the `-f config_file` its configuration file is `/etc/newsyslog.conf`
- the granularity of the rotating time is determined on how often *newsyslog* is run

BSD newsyslog configuration file

- *newsyslog* configuration file is `/etc/newsyslog.conf`. It consists of lines, each line referring to a file to be rotated
- blank lines are ignored, lines starting with # are treated as comments
- each line has the following format

```
logfile owner:group  mode count size when  flags
```

openBSD newsyslog configuration file

- **logfile** the log file being rotated
- **owner:group** (optional) the owner and group of the rotated file
- **mode** permissions of the rotated file
- **count** how many rotated files to keep
- **size** rotate when this size is reached (* or 0 to rotate only on time)
- **when** rotate when this age, or date is reached (* to rotate only on size)
- **flags** Z and J for compressing files, C to create the file if it does not exists, M for monitoring (a username or email address should be included as the next field to be notified the file has been rotated), ...

sample openBSD newsyslog configuration file

```
#       $OpenBSD: newsyslog.conf,v 1.29 2011/04/14 20:32:34 sthen Exp $
#
# configuration file for newsyslog
#
# logfile_name          owner:group     mode count size when  flags
/var/cron/log           root:wheel      600  3     10   *     Z
/var/log/aculog         uucp:dialer     660  7     *    24    Z
/var/log/authlog        root:wheel      640  7     *    168   Z
/var/log/daemon                         640  5     30   *     Z
/var/log/lpd-errs                       640  7     10   *     Z
/var/log/maillog                        600  7     *    24    Z
/var/log/messages                       644  5     30   *     Z
/var/log/secure                         600  7     *    168   Z
/var/log/wtmp                           644  7     *    $W6D4 B
/var/log/xferlog                        640  7     250  *     Z
/var/log/ppp.log                        640  7     250  *     Z
/var/log/pflog                          600  3     250  *     ZB "pkill -HUP -u root -U root -t - -x pflog
```

linux logrotate

- logrotate takes care of rotating, compressing, removing,... of log files in linux systems
- it is usually run daily through cron
- *logrotate* has its configuration file `/etc/logrotate.conf`
  - it has some global options which can be overriden by per-file options
  - specific options for some logfile can be specified in the format

    ```
    logfile  {
      options
    }
    ```

  - additional specific file configurations can be put in the *logrotate* configuration directory, specified in *logrotate* configuration file (typically `/etc/logrotate.d`)

sample linux logrotate configuration file

```
# see "man logrotate" for details
# rotate log files weekly
weekly

# keep 4 weeks worth of backlogs
rotate 4

# create new (empty) log files after rotating old ones
create

# uncomment this if you want your log files compressed
#compress

# packages drop log rotation information into this directory
include /etc/logrotate.d

# no packages own wtmp, or btmp -- we'll rotate them here
/var/log/wtmp {
    missingok
    monthly
    create 0664 root utmp
    rotate 1
}

/var/log/btmp {
    missingok
    monthly
    create 0660 root utmp
    rotate 1
}
```

sample /etc/logrotate.d/apache

```
/var/log/apache2/*.log {
        weekly
        missingok
        rotate 52
        compress
        delaycompress
        notifempty
        create 640 root adm
        sharedscripts
        postrotate
                /etc/init.d/apache2 reload > /dev/null
        endscript
        prerotate
                if [ -d /etc/logrotate.d/httpd-prerotate ]; then \
                        run-parts /etc/logrotate.d/httpd-prerotate; \
                fi; \
        endscript
}
```

Starting and stopping system services. boot scripts

system V init scripts
linux
solaris
openbsd
freebsd
netbsd

boot process

- We've seen before the general boot procedure for a O.S.
- The kernel, after been loaded and having performed its initialization routines, starts the first user program in the system: `init`
- `init` checks the file `/etc/inittab` for the *runlevel* the system must be brought to, and after running the *scripts* in `/etc/rcS.d` it invokes the ones in`/etc/rcN.d` (where N is the *runlevel*) which will be starting the available services in that installation.

boot process

- A similar process is performed when the system is brought down to stop all the services in an ordenated way
- The system has more than one predetermined configuration
  - each of them with its particular set of running services

Init system: Runlevels

- Runlevels are configurations of services running in the system. When entering a runlevel, some services are started and some are stopped.
- We have 7 runlevels predefined, although that may be different on different systems. These runlevels typically are
  - 0: system halted.
  - 1: single-user mode
  - 2: multi-user mode
  - 3: multi-user mode graphic environment
  - 4: Available.
  - 5: Available
  - 6: Reboot
- To change the runlevel we use `telinit`
- To get the current runlevel, we use `runlevel`

Init system: Runlevels

- There is a directory where all the scripts necessary to start and stop services reside. It is usually `/etc/init.d`
- These scripts accept several parameters, (typically `start`, `stop`, `restart ...`), to indicate which action we want to perform. This also allows us to start or stop a service at anytime
- For each runlevel there's a directory `rcN.d` (`rc0.d`, `rc1.d`, `rc2.d...`) in `/etc/`, o in `/etc/rc.d`. This directory contains links to the actual scripts in `/etc/init.d`. This links are in the form

  `S00exim`

  `K20ssh`

Init system: Runlevels

- The first letter (S or K) indicates whether the service is to be started (**S**) or stopped (**K**) in that runlevel.
- The number defines the order in which services are started (or stopped).
- If we want to perform actions at system startup or shutdown we only need to place an adequate script in the corresponding runlevel. (more correctly, we put the script in `/etc/init.d` and create the adequate links in `/etc/rcN.d`
- `runlevel` *newlevel* changes the runlevel to *newlevel*. For example, to switch to single-user mode, we'd use

  ```
  # telinit 1
  ```

System V init system

- System V init system was the init system of choice in previous solaris and linux systems
- From version 10, solaris has adopted the *smf* (Services Management Facilities) implementation, that we'll see later on
- Many linux distros have (unfortunately) adopted *systemd*, although there still some that use System V init system, even with some enhancements

devuan insserv

- devuan insserv is an extension to the system V style initialization files
- Provides a more sophisticated control of the dependencies among the initialization scripts
- Allows for parallel execution of the scripts
- Introduced in debian *squeeze*
- *insserv* takes care of arranging for the scripts to be run at boot (or shutdown) time when appropriate
- The links to the *scripts* in `/etc/init.d` from the corresponding */etc/rcN.d* directory are also automatically generated
- For example, to run *anacron* in the default runlevels

```
# insserv -d anacron
```

- To get *anacron* not executed at system startup (also the links will be removed)

devuan insserv

- In order to use *insserv* the initialization *scripts* must have a specific format
- *boot_facilities* descriptions can be seen in `/etc/insserv.conf` or in the files at `/etc/insserv.conf.d/`
- A sample script header. To specify dependencies we can also use *$all* o *$null*

```
### BEGIN INIT INFO
# Provides:          boot_facility_1 [ boot_facility_2 ...]
# Required-Start:    boot_facility_1 [ boot_facility_2 ...]
# Required-Stop:     boot_facility_1 [ boot_facility_2 ...]
# Should-Start:      boot_facility_1 [ boot_facility_2 ...]
# Should-Stop:       boot_facility_1 [ boot_facility_2 ...]
# X-Start-Before:    boot_facility_1 [ boot_facility_2 ...]
# X-Stop-After:      boot_facility_1 [ boot_facility_2 ...]
# Default-Start:     run_level_1 [ run_level_2 ...]
# Default-Stop:      run_level_1 [ run_level_2 ...]
# X-Interactive:     true
# Short-Description: single_line_description
# Description:       multiline_description
### END INIT INFO
```

devuan insserv

- Example of a simple *script* header for a system that gets started in runlevels 2,3,4,5 y S and stopped at runlevels 0, y 6

```
### BEGIN INIT INFO
# Provides:          cortafuegos
# Required-Start:    $network
# Required-Stop:
# Default-Start:     2 3 4 5 S
# Default-Stop:      0 6
# Short-Description: Inicia o para el cortafuegos
# Description:       Rechaza conexiones de fuera del departamento
#                    que venga a un puerto distinto de 80
### END INIT INFO
```

- The easiest way to get a script executed at system startup is to have it included in `/etc/rc.local`

devuan insserv

- alternatively we can use the command `update-rc.d` to enable or disable services
- we can force the system into using the legacy mode (manual ordering of the scripts) by creating a file named `.legacy-bootordering` in the `/etc/init.d` directory

```
# touch  /etc/init.d/.legacy-bootordering
```

fedora systemctl

- fedora linux (together with its sister distro, red hat) were among the first distros to adopt *systemd* as its initialization system
- system services in fedora are controlled by *systemd* via the command `systemctl`
- the command `systemctl` allows us to start, stop, restart, ... what it calls *units*
- a *unit* can be of type *service* (daemon or process), *socket* (connection), *device*, *mount* (mount point) ...
- examples.
  - To start telnet

    ```
    # systemctl start telnet.socket
    ```
  - To stop telnet

    ```
    # systemctl stop telnet.socket
    ```

fedora systemctl examples

- examples.
  - To enable telnet at system startup

    ```
    # systemctl enable telnet.socket
    ```
  - to disable telnet at system startup

    ```
    # systemctl disable telnet.socket
    ```
  - to start the sshd service

    ```
    # systemctl start sshd.service
    ```
  - to disable sshd at system startup

    ```
    # systemctl disable sshd.service
    ```
  - to list what units we have available

    ```
    # systemctl list-units
    ```

fedora startup services

- fedora still supports a legacy system V style services, although its `/etc/init.d` directory is not fully populated.
- it is only for a few system services or for the user to add their own
- we can add our own scripts to the /etc/init.d directory to have them started or stopped at boot time
- our scripts must understand the parameters `start`, `stop`, `status`, `restart` ...
- the services can then be managed with `ntsysv`, `chkconfig` or `system-config-services` and started or stopped with the service command
- however for the services to be managed this way, either hints to chkconfig must be provided or a LSB style stanza has to be included

fedora startup services

- example of chkconfig hints stating that the script can affect runlevels 2,3,4, and 5, and the links will be S20 or K80

```
# chkconfig: 2345 20 80
# description: This lines will appear as the description \
#              of the service this script provides.
```

- LSB style stanza

```
### BEGIN INIT INFO
# Provides: foo
# Required-Start: bar
# Defalt-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Description: Foo init script
### END INIT INFO
```

debian and systemd

- As of version 8 (Jessie) debian has adopted (as other linux distros) *systemd* as its default procedure for system initialization
- We still can use the scripts in `/etc/init.d` to start and stop services (which they do via the command `systemctl`)
- the command `systemctl` allous us to start, stop, enable and disable services
- The configuration of systemd is represented by links in the /etc/systemd directory
- The switch to *systemd* has not been well received by many, if fact a fork of the debain project (without systemd) has been made; devuan (www.devuan.org)

ununtu and systemd

- with debian's move to systemd, ubuntu has also adopted systemd as its init system
- ubuntu still retains system V style initialization scripts: a fully populated `/etc/init.d`
- services can be started and stopped the usual system V way `/etc/init.d/service-name start;` `/etc/init.d/service-name stop`
- services cal also be started and stopped with the *service* command (`service service-name start; service service-name stop`
- both ways `/etc/init.d ...` and `service ...` are in turn calls to *systemctl* that has *systemd* actually start and/or stop the services.

solaris smf

- prior to solaris 10, services were started with the standard System V initialization (*inittab + /etc/init.d scripts*)
- as this scripts run sequentially
  - the boot process can take long
  - service dependencies must be known to run the scripts in the appropiate order
- this has been addressed with the new *smf* (Service Management Facilities)
- services can also be configured the *old way*. These are referred as legacy services

solaris boot process

- After control of the system is passed to the kernel, the system begins the last stage of the boot process: the `init` stage
- `init` reads the file `/etc/default/init` to set variables that will get passed to its descendant processes, and reads the file `/etc/inittab` which tells which programs to start
- it starts the `svc.startd` daemon which is responsible for starting and stopping other system services such as mounting file systems, configuring network devices ... `svc.startd` will also execute legacy run control (rc) scripts

solaris runlevels

- starting with version 10, solaris doesn't use runelvels as such
  - the runlevel can still be got with the **who -r** command
  - the run level can still be changed with `/sbin/init`. for example

    ```
    # /sbin/init 1
    ```

    brings the machine to single user mode
  - it introduces the concept of *milestones*. A milestone is a special type of service that represents a group of services

solaris milestones

- there are 3 milestone predefined
  - `svc:/milestone/single-user` (more or less) equivalent to `init S`
  - `svc:/milestone/multi-user` (more or less) equivalent to `init 2`
  - `svc:/milestone/multi-user-server` (more or less) equivalent to `init 3`
- there are also defined the milestones '*none*' (maintenance mode, none services being run), and '*all*', all services being run (default system run). For example:

  ```
  # svcadm milestone none
  ```

  enters system maintenance mode, and

  ```
  # svcadm milestone all
  ```

  restores normal system functioning

managing services

- the basic commands to manage services are
  - **svcs** list all the services and their status. It also lists the legacy services
  - **svcs -l service_name** gets details on *service_name*
  - **svcadm enable service_name** starts *service_name*. *service_name* will also get started the next reboot
  - **svcadm disable service_name** stops *service_name*. *service_name* will not get started the next reboot
  - **svcadm restart service_name** restarts *service_name*
  - **svcadm disable -t service_name** stops *service_name*. *service_name* will get started the next reboot if it was supposed to do so
  - **svcprop service_name** displays *service_name* properties

managing services

- services are defined by XML files in the directory `/var/svc/manifest`
- the XML file of a service contains references to scripts that start and stop the service
- these scripts are in `/lib/svc/method`

openBSD boot

- when the system starts up, after the kernel has performed its initialization routines, the `init` process is created. It runs the script `/etc/rc`
- `/etc/rc`
  - checks filesystems
  - reads configuration variables from `/etc/rc.conf` and then from `/etc/rc.conf.local`
  - configures the network with `/etc/netstart`
  - runs `/etc/rc.local`
  - runs local and package scripts in `/etc/rc.d`
- when the system goes down it executes the script `/etc/rc.shutdown`

starting services in openBSD

- to start a service provided in openBSD we set the appropiate variable in `/etc/rc.conf.local` (`/etc/rc.conf` should be left untouched and used as a guide). for example the following line in `/etc/rc.conf.local`

  ```
  sshd_flags=""
  ```

  gets `sshd` started at system startup whereas

  ```
  sshd_flags=NO
  ```

  does not

starting services in openBSD

- to start services not provided with openbsd we can
  a) start them directly form `/etc/rc.local`
  b) place a script in `/etc/rc.d` and specify the name of the script in the variable *pkg_scripts* in `/etc/rc.conf.local`
     - the standard scripts accept the parameters *start*, *stop*, *restart*, *reload*, *check*
     - most of these scripts just call to `/etc/rc.d/rc.subr` that takes care of processing the parameters. Sample *sshd* script in openbsd

```
#!/bin/sh
#
# $OpenBSD: sshd,v 1.1 2011/07/06 18:55:36 robert Exp $

daemon="/usr/sbin/sshd"

. /etc/rc.d/rc.subr

rc_cmd $1
```

openBSD securelevels

- bsd systems do not have runlevels
- they have what it is called *securelevels*
- when the system boots it is in *securelevel=0*, also called *insecure mode*, In this mode
  - all devices can be accesed as dictated by their permissions
  - system file flags may be changed with **chflags**
  - kernel modules may be loaded or unloaded

openBSD securelevels

- the file **/etc/rc.securelevel** dictates which securelevel must be reached when the system boot has completed
- the usual multiuser environment is *securelevel=1*. At this level
  - securelevel may not be lowered
  - raw devices of mounted file systems are read-only, independent of their permissions
  - kernel modules may not be loaded or unloaded
  - certain system variables may not be changed
  - `/dev/mem` and `/dev/kmem` are not writable

openBSD securelevels

- the system also has another securelevel, *securelevel=2* o highly secure mode
- This level is like *securelevel=1* and also
  - pf and nat rules can no be changed
  - all raw disk devices are read-only
  - time can not be changed backwards or close to overflow

openBSD securelevels

- there's also *securelevel=-1* which is exactly as *securelevel=0*, execpt that **init** will not try to raise the securelevel
- to raise the securelevel the *kern.securelevel* variable must be set with **sysctl**
- only **init** may lower the securelevel
  - to enter *securelevel=0*, from a securelevel, SIGTERM must be sent to **init**

freeBSD services

- in freeBSD the scripts to start and stop services are located in the directory `/etc/rc.d` (`/usr/local/etc/rc.d` for services provided by packages)
- system services are controlled through variables defined in `/etc/rc.conf`. We can get the name of the variable controlling the execution of the service with the `rcvar` option of the `service` command

```
# service sshd rcvar
# sshd
sshd_enable="YES"
#    (default="")
```

- we can get the system to start sshd at boot with `sshd_enable="YES"` in `/etc/rc.conf`

freeBSD services

- if the service is enabled we can start (or stop) it with

  ```
  # service sshd start
  ```

- if the service is not enabled we can start it with

  ```
  # service sshd onestart
  ```

- to determine if it is running we use the `status` parameter to the `service` command

  ```
  # service sshd status
  ```

freeBSD services

- should we want some task be performed at boot or shutdown time we can simply include it in the
  - `/etc/rc.local` to get it run at boot time
  - `/etc/rc.shutdown` to get it run at shutdown time
- we can also write a script and have it handled with `rc.subr` through the variables in `/etc/rc.conf`
- the script should be placed in `/etc/rc.d` or in `/usr/local/etc/rc.d`
- an outline of creating rc scripts can be found in

  `http://www.freebsd.org/doc/en_US.ISO8859-1/articles/rc-scripting/`

freeBSD securelevels I

- like openBSD freeBSD defines several securelevels. freeBSD defines 5
  - -1 Permanently insecure mode. This is the default initial value
  - 0 Insecure mode: Immutable and append-only flags may be turned off. All devices may be read or written subject to their permissions
  - 1 Secure mode: The system immutable and system append-only flags may not be turned off; disks for mounted file systems, `/dev/mem` and `/dev/kmem` may not be opened for writing. Kernel modules may not be loaded or unloaded.
  - 2 Highly secure mode: Same as secure mode, plus disks may not be opened for writing, except by the *mount* command whether mounted or not. Kernel time changes are restricted to less than, or equal to, one second.

freeBSD securelevels II

3 Network secure mode: same as highly secure mode, plus IP packet filter rules cannot be changed and `pf` configuration cannot be adjusted.

- the *securelevel* can be raised by root but it can not be lowered
- it can be modified by changing the *kern.securelevel* variable
- this can be done with the `sysctl` command or by specifying `kern.securelevel=value` in the `/etc/rc.conf` file

netBSD services

- in netBSD the scripts to start and stop services are located in the directory `/etc/rc.d`.
- Scripts provided by packages (for example, slim) are provided typically in `/usr/pkg/share/examples/rc.d` and should be copied to `/etc/rc.d`
- system services are controlled through variables defined in `/etc/rc.conf`. We can get the name of the variable controlling the execution of the service with the `rcvar` option of the `service` command or pass rcvar to the appropiate `/etc/rc.d` script

```
aso22-3 # service inetd rcvar
# inetd
$inetd="YES"
aso22-3 # /etc/rc.d/inetd rcvar
# inetd
$inetd="YES"
```

- we can get the system to start inetd at boot with `inetd=YES` in `/etc/rc.conf`

netBSD services

- if the service is enabled we can start (or stop) it with

```
# service inetd start
# /etc/rc.d/inetd start
```

- if the service is not enabled we can start it with

```
# service inetd onestart
# /etc/rc.d/inetd onestart
```

- to determine if it is running we use the `status` parameter to the `service` command

```
# service sshd status
# /etc/rc.d/inetd status
```

netBSD services

- should we want some task be performed at boot or shutdown time we can simply include it in the
  - `/etc/rc.local` to get it run at boot time
  - `/etc/rc.shutdown` to get it run at shutdown time
- we can also write a script and have it handled with `rc.subr` through the variables in `/etc/rc.conf`
- the script should be placed in `/etc/rc.d`

netBSD securelevels I

- like openBSD and freeBSD netBSD defines several securelevels. freeBSD defines 4
  - -1 **Permanently insecure mode:** This is the default initial value
  - 0 **Insecure mode:** The init process (PID 1) may not be traced or accessed by ptrace(2), systrace(4), or procfs. Immutable and append-only flags may be turned off. All devices may be read or written subject to their permissions
  - 1 **Secure mode:** All effects of securelevel 0. /dev/mem and /dev/kmem may not be written to. Raw disk devices of mounted file systems are read-only. Immutable and append-only file flags may not be removed. Kernel modules may not be loaded or unloaded. The net.inet.ip.sourceroute sysctl(8) variable may not be changed. Adding or removing sysctl(9) nodes is denied. The RTC offset may not be changed. Set-id coredump settings may not be altered. Attaching the IP-based kernel debugger, ipkdb(4), is not allowed. Device

netBSD securelevels II

'pass-thru' requests that may be used to perform raw disk and/or memory access are denied. iopl and ioperm calls are denied. Access to unmanaged memory is denied

2 **Highly secure mode:** All effects of securelevel 1. Raw disk devices are always read-only whether mounted or not. New disks may not be mounted, and existing mounts may only be downgraded from read-write to read-only. The system clock may not be set backwards or close to overflow. Per-process coredump name may not be changed. Packet filtering and NAT rules may not be altered

netBSD securelevels I

- the *securelevel* can be raised by root but it can not be lowered
- it can be modified by changing the *kern.securelevel* variable
- this can be done with the `sysctl` command or by specifying `kern.securelevel=value` in the `/etc/rc.conf` file