86

AIX update

December 2002

In this issue

AIX Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38342
From USA: 01144 1635 38342
E-mail: trevore@xephon.com

North American office

Xephon
PO Box 350100
Westminster, CO 80035-0100
USA
Telephone: 303 410 9344

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1999 issue, are available separately to subscribers for £16.00 ($24.00) each including postage.

AIX Update on-line

Code from *AIX Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at http://www.xephon.com/aix; you will need to supply a word from the printed issue.

Editors

Trevor Eddolls

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

Printed in England.

Where's that filesystem gone?

Following on from the article I wrote that appeared in Issue 69 (July 2001) of *AIX Update* entitled *Tidy up before you go!*, there is another (potential) pitfall in AIX whereby a newly-defined filesystem can overlay a previously defined filesystem, making the earlier filesystem 'disappear'. For example, if you have the following filesystems on a machine:

```
/
/usr
/var
/tmp
/home
/apps/data/a/b/c
```

and you define a new filesystem:

```
/apps/data/a/b
```

and mount that, then the data in */apps/data/a/b/c* will 'disappear'. There is nothing that will warn you about it either. I have seen this a few times, sometimes by mistake (doh!), and sometimes when doing an **importvg**. The **importvg** function can put filesystem entries into */etc/filesystems* in a different order from the one in which they were exported.

The following program was written to help catch these situations. Run as part of the 'tidy up' suite once a day, it will notify you of problems with filesystem overlays. The following example shows a sample */etc/filesystems* file (which is processed top to bottom), having four potential errors, which are shown in the output, also below:

```
/:
     dev       = /dev/hd4
     vfs       = jfs
     log       = /dev/hd8
     mount     = automatic
     check     = false
     type      = bootfs
     vol       = root
     free      = true
```

```
/home:
     dev       = /dev/hd1
     vfs       = jfs
     log       = /dev/hd8
     mount     = true
     check     = true
     vol       = /home
     free      = false

/usr:
     dev       = /dev/hd2
     vfs       = jfs
     log       = /dev/hd8
     mount     = automatic
     check     = false
     type      = bootfs
     vol       = /usr
     free      = false

/var:
     dev       = /dev/hd9var
     vfs       = jfs
     log       = /dev/hd8
     mount     = automatic
     check     = false
     type      = bootfs
     vol       = /var
     free      = false

/tmp:
     dev       = /dev/hd3
     vfs       = jfs
     log       = /dev/hd8
     mount     = automatic
     check     = false
     vol       = /tmp
     free      = false

/usr/local/apps/data/a/b/c/d:
     dev       = "/dev/abcd"
     vfs       = jfs
     mount     = false
     log       = /dev/loglvØØ

/usr/local/apps/data/a:
     dev       = "/apps/data/a"
     vfs       = nfs
     nodename  = othermc
     mount     = false

/usr/local/apps/data/a/b/c:
     dev       = "/apps/data/a/b/c"
```

```
     vfs      = nfs
     nodename = othermc
     mount    = true

/usr/local/apps/data/a/b:
     dev      = "/dev/ab"
     vfs      = jfs
     mount    = true
     log      = /dev/loglvØØ
```

This will produce the following output:

```
Checking machine TESTMACH for filesystems errors :
================================================
ERROR - jfs filesystem '/usr/local/apps/data/a/b' has a subfilesystem
entry already (/usr/local/apps/data/a/b/c/d)

ERROR - jfs filesystem '/usr/local/apps/data/a/b' has a subfilesystem
entry already (/usr/local/apps/data/a/b/c)

ERROR - nfs filesystem '/usr/local/apps/data/a/b/c' has a subfilesystem
entry already (/usr/local/apps/data/a/b/c/d)

WARNING - nfs filesystem '/usr/local/apps/data/a' has a subfilesystem
entry already but is mount=false (/usr/local/apps/data/a/b/c/d)
```

FILESYS_CHECK.PL

```
#!/usr/bin/perl -w
$|=1;
#
# Program to trap potential filesystem problems regarding overlays
#
use strict;
#use Data::Dumper;
use Sys::Hostname;
my $hostname=hostname();
my %fshash;                  # Hash to hold filesystem_name/vfs/mount
options
my @fserrors;                # Array of filesystems with (potential)
errors
my $fsname;
my $fs;
print "\nChecking machine $hostname for filesystems errors :\n";
print "================================================\n";
open(FS,"/etc/filesystems") or die "Cannot open /etc/filesystems!\n";
while(<FS>) {
  next if /^\*/;
  s/^\s+//;
  s/\s+$//;
  SWITCH: {
```

```
    /^\/.*:$/ and do {
      $fsname=(split (/:/, $_))[0];
      foreach $fs (keys %fshash) {
        if ( grep /$fsname/, $fs ) {
          push(@fserrors,$fsname);
          push(@fserrors,$fs);
        }
      }
      last SWITCH;
    };
    /^mount/ and do {
      $fshash{$fsname}{"mount"}=(split(/=/, $_))[1];
      $fshash{$fsname}{"mount"}=~s/^\s+//;
      last SWITCH;
    };
    /^vfs/ and do {
      $fshash{$fsname}{"vfs"}=(split(/=/, $_))[1];
      $fshash{$fsname}{"vfs"}=~s/^\s+//;
      last SWITCH;
    };
  }
}
close FS;
if ( scalar(@fserrors) eq 0 ) {
  print "None\n\n";
} else {
   while ( scalar(@fserrors) gt 0 ) {
    $fsname=pop(@fserrors);
    $fs=pop(@fserrors);
    if ( $fshash{$fs}{'mount'} =~ "true" ) {
      print "ERROR - ".$fshash{$fs}{'vfs'}." filesystem '$fs' has a
subfilesystem entry already ($fsname)\n\n";
    } else {
      print "WARNING - ".$fshash{$fs}{'vfs'}." filesystem '$fs' has a
subfilesystem entry already but is mount=false ($fsname)\n\n";
    }
  }
}
#!/usr/bin/perl -w
$|=1;
#
# Program to trap potential filesystem problems regarding overlays
#
use strict;
#use Data::Dumper;
use Sys::Hostname;
my $hostname=hostname();
my %fshash;                  # Hash to hold filesystem_name/vfs/mount
options
my @fserrors;                # Array of filesystems with (potential)
errors
my $fsname;
```

```
my $fs;
print "\nChecking machine $hostname for filesystems errors :\n";
print "=================================================\n";
open(FS,"/etc/filesystems") or die "Cannot open /etc/filesystems!\n";
while(<FS>) {
  next if /^\*/;
  s/^\s+//;
  s/\s+$//;
  SWITCH: {
    /^\/.*:$/ and do {
      $fsname=(split (/:/, $_))[0];
      foreach $fs (keys %fshash) {
        if ( grep /$fsname/, $fs ) {
          push(@fserrors,$fsname);
          push(@fserrors,$fs);
        }
      }
      last SWITCH;
    };
    /^mount/ and do {
      $fshash{$fsname}{"mount"}=(split(/=/, $_))[1];
      $fshash{$fsname}{"mount"}=~s/^\s+//;
      last SWITCH;
    };
    /^vfs/ and do {
      $fshash{$fsname}{"vfs"}=(split(/=/, $_))[1];
      $fshash{$fsname}{"vfs"}=~s/^\s+//;
      last SWITCH;
    };
  }
}
close FS;
if ( scalar(@fserrors) eq 0 ) {
  print "None\n\n";
} else {
   while ( scalar(@fserrors) gt 0 ) {
    $fsname=pop(@fserrors);
    $fs=pop(@fserrors);
    if ( $fshash{$fs}{'mount'} =~ "true" ) {
      print "ERROR - ".$fshash{$fs}{'vfs'}." filesystem '$fs' has a
subfilesystem entry already ($fsname)\n\n";
    } else {
      print "WARNING - ".$fshash{$fs}{'vfs'}." filesystem '$fs' has a
subfilesystem entry already but is mount=false ($fsname)\n\n";
    }
  }
}
```

Phil Pollard
Unix and TSM Administrator (UK)

Syslog – the AIX system logger

A critical part of the system manager's job is that of monitoring the system. The most portable facility that supports this is the syslog system. Available since the earliest releases of BSD Unix, it is now supported on most versions of Unix including AIX. Syslog enables you to sort messages by level and facility. Levels indicate various degrees of severance of logged event (eg warning, error, emergency) whereas facilities are considered service areas (eg printing, e-mail, network). Syslog has always offered not just local logging to files but also remote logging over the network via standard protocols to heterogeneous systems.

SYSLOG ARCHITECTURE

The syslog system consists of the following components:

- Message format specification:

  /usr/include/sys/syslog.h

  Syslog messages are encoded as ASCII strings. Message strings are created throughout the AIX system. Messages are created at one of a set of possible levels; by setting a threshold, one can direct all messages at or above a certain level to a given destination.

- A set of system utilities for creating messages using C API:

  syslog(3),syslog_r(3), logger(1)

  Most users create syslog messages using one of the standard interfaces to syslog. The C-code interface contained in the /usr/*ccs/lib/libc.a* and the thread-safe interface contained in the */usr/ccs/lib/libc_r.a* are similar in functionality to the standard **printf** interface. The command line utility **logger** can be used from the command line or a shell script.

- A set of locations from which messages can be read:

  /dev/log, UDP port 514.

Each syslog message must be directed to one of a number of communication paths that are read by syslod daemon. These communication channels include the socket file */dev/log,* which is used to route locally-produced messages, and UDP port 514, used to deliver messages from remote hosts.

- A set of locations to which messages are directed: files, users, etc.

  This set of locations may include sockets, files, consoles, and logged-on users. Messages can be directed to one or more of these destinations.

- Rules that govern the distribution of the messages to various destinations: */etc/syslog.conf*.

  This set of configuration rules determines how messages are logged and the locations to which they are logged. The rules are set by the system administrator in the file */etc/syslog.conf*.

SYSLOG MESSAGE FORMAT

One of the unique characteristics of syslog is that every message is created and logged in the form of a plain text ASCII string. This string consists of the following fields:

- A message priority.

  Priority is an ASCII integer encoding of an 8-bit quantity. This quantity is a combination of a 3-bit field (bits 0 to 2) used to designate message priority and a 5-bit field (bits 3 to 7) used for message facility. Thus, message priority level can have eight possible values, and message facility up to 32 possible values.

  Message priorities are defined as an ordered list. If one has set a threshold at a given priority, one will receive all messages at this or any higher priority. Thus, if the syslog has been configured to log all messages tagged LOG_WARNING, you will also log all messages of higher priorities, such as LOG_ERR, LOG_CRIT. The defined

message priorities, from highest to lowest, are as follows:

– LOG_EMERG – 0 – system is unusable
– LOG_ALERT – 1 – action must be taken immediately
– LOG_CRIT – 2 – critical conditions
– LOG_ERR – 3 – error conditions
– LOG_WARNING – 4 – warning conditions
– LOG_WARNING – 5 – normal but significant condition
– LOG_INFO – 6 – informational
– LOG_DEBUG – 7 – debug-level messages.

The message facility identifies the originating subsystem of the message. Facility indications are defined in */usr/include/sys/syslog.h*; some are reserved for the OS and others are available for users and application developers. The following message facility tags are defined:

– LOG_KERN – (0<<3) – kernel messages
– LOG_USER – (1<<3) – random user-level messages
– LOG_MAI – (2<<3) – mail system
– LOG_DAEMON – (3<<3) – system daemons
– LOG_AUTH – (4<<3) – security/authorization messages
– LOG_SYSLOG – (5<<3) – messages generated by syslogd
– LOG_LPR – (6<<3) – line printer subsystem
– LOG_NEWS – (7<<3) – news subsystem
– LOG_UUCP – (8<<3) – uucp subsystem
– LOG_CRON – (9<<3) – clock daemon
– LOG_AUTHPRIV – (10<<3) – security/authorization messages

  - LOG_LOCAL0 – (16<<3) – reserved for local use
  - LOG_LOCAL1 – (17<<3) – reserved for local use
  - LOG_LOCAL2 – (18<<3) – reserved for local use
  - LOG_LOCAL3 – (19<<3) – reserved for local use
  - LOG_LOCAL4 – (20<<3) – reserved for local use
  - LOG_LOCAL5 – (21<<3) – reserved for local use
  - LOG_LOCAL6 – (22<<3) – reserved for local use
  - LOG_LOCAL7 – (23<<3) – reserved for local use.
- A timestamp

  This is an ASCII-encoded date/time string.
- The message string

  This is the user-supplied message to be logged.

  The logged message is the ASCII string that is written by the syslogd daemon to the user's terminal, to the console, or to a log file. When the message is displayed the message priority is omitted; this information is used by syslogd to direct the syslog message according to the configuration rules defined in the file /etc/*syslog.conf*. The first field to be displayed is therefore the timestamp, followed by the name of the originating system. This is the local system hostname if the message was generated on a local system or a remote system hostname communicating over the UDP socket. The message string is the actual string of the message; useful data such as process ID and a message prefix may be inserted in this message using the **openlog()** function.

GENERATION OF SYSLOG MESSAGES USING SYSLOG(3) AND LOGGER(1)

The facilities offered by AIX for the purpose of creating syslog messages are a set of C APIs containing the functions **openlog(), syslog(), closelog(),** and **setlogmask()**. The logger command

/usr/bin/logger provides similar functionality from a command line or shell script.

The syslog subroutine writes messages onto the system log maintained by the syslogd daemon. A typical usage example is:

```
syslog(LOG_CRIT, "filesystem is full");
```

The message is similar to the **printf fmt** *string*; the difference is that *%m* is replaced by the current error message obtained from the **errno** global variable. A trailing new-line can be added to the message if needed.

Messages are read by the syslogd daemon and written to the system console or log file, or forwarded to the syslogd daemon on the appropriate host.

If special processing is required, the **openlog** subroutine can be used to initialize the log file. The following is a typical example:

```
openlog("Disk space monitoring facility", LOG_PID|LOG_CONS, LOG_LOCAL0);
```

Messages are tagged with codes indicating the type of priority for each. A priority is encoded as a facility, which describes the part of the system generating the message, and as a level, which indicates the severity of the message.

If the syslog subroutine cannot pass the message to the **syslogd** command, it writes the message on the */dev/console* file, provided the **LOG_CONS** option is set.

The **closelog()** subroutine closes the log file.

The **setlogmask(MaskPriority)** subroutine uses the bit mask in the MaskPriority parameter to set the new log priority mask and returns the previous mask.

The **LOG_MASK** and **LOG_UPTO** macros in the */usr/include/sys/syslog.h* file are used to create the priority mask. Calls to the syslog subroutine with a priority mask that does not allow logging of that particular level of message causes the subroutine to return without logging the message.

The **logger** command provides an interface to the syslog

subroutine, which writes entries to the system log. A message variable can be specified on the command line, which is logged immediately, or a file variable is read and each line of the file variable is logged. If you specify no flags or variables, the **logger** command will wait for you to enter a message from standard input. The **logger** command has the following command line arguments:

- **-f** File – logs the specified file variable. If the message variable is specified, this flag is ignored.
- **-i** – logs the process ID of the logger process with each line.
- **-p** Priority – enters the message with the specified priority. The priority parameter may be a number or a facility level priority specifier.
- **-t** Tag – marks every line in the log with the specified tag parameter.
- Message – indicates the message to log. If this variable is not specified, the **logger** command logs either standard input or the file specified with the **-f** File flag.

To log a message indicating a system reboot, enter:

```
logger System rebooted
```

To log a message contained in the /tmp/msg1 file, enter:

```
logger -f /tmp/msg1
```

To log the daemon facility critical level messages, enter:

```
logger -pdaemon.crit
```

SYSLOGD DAEMON

When the AIX is booted the syslogd daemon is started by the script */etc/rc.tcpip*, which is executed by init process. The syslogd daemon reads a datagram socket and sends each message line to a destination described by the */etc/syslog.conf* configuration file. The syslogd daemon reads the configuration

file when it is activated and when it receives a hang-up signal. The syslogd daemon creates the */etc/syslog.pid* file, which contains a single line with the command process ID used to end or reconfigure the syslogd daemon.

A terminate signal sent to the syslogd daemon ends the daemon. The syslogd daemon logs the end-signal information and terminates immediately.

The syslogd daemon has the following command line arguments:

- **-d** – turns on debugging.
- **-f** ConfFile – specifies an alternative configuration file.
- **-m** MarkInterval – specifies the number of minutes between the **mark** command messages. If you do not use this flag, the **mark** command sends a message with **LOG_INFO** priority sent every 20 minutes. This facility is not enabled by a selector field containing an ***** (asterisk), which selects all other facilities.
- **-s** – specifies that a 'shortened' message should be forwarded to another system (if it is configured to do so) for all the forwarding **syslog** messages generated on the local system.
- **-r** – suppresses logging of messages received from remote hosts.

FORMAT OF SYSLOGD CONFIGURATION FILE

The configuration file informs the syslogd daemon where to send a system message, depending on the message's priority level and the facility that generated it.

If you do not use the **-f** flag, the syslogd daemon reads the default configuration file, the */etc/syslog.conf* file.

The syslogd daemon ignores blank lines and lines beginning with a #.

Lines in the configuration file for the syslogd daemon contain a

selector field and an action field, separated by one or more tabs.

The selector field names a facility and a priority level. Separate facility names with a , (comma). Separate the facility and priority-level portions of the selector field with a . (period or full stop). Separate multiple entries in the same selector field with a ; (semicolon). To select all facilities, use an * (asterisk).

The action field identifies a destination (file, host, or user) to receive the messages. If routed to a remote host, the remote system will handle the message as indicated in its own configuration file. To display messages on a user's terminal, the destination field must contain the name of a valid, logged-in, system user.

An optional rotation field, separated by one or more tabs, has been introduced in AIX 5.1. The rotation field identifies how rotation is used. If the action field is a file, then rotation can be based on size or time, or both. One can also compress and/or archive the rotated files.

Facilities

Use the following system facility names in the selector field:

- **kern** – kernel
- **user** – user level
- **mail** – mail subsystem
- **daemon** – system daemons
- **auth** – security or authorization
- **syslog** – syslogd daemon
- **lpr** – line-printer subsystem
- **news** – news subsystem
- **uucp** – uucp subsystem
- ***** – all facilities.

Priority levels

Use the following message priority levels in the selector field. Messages of the specified priority level and all levels above it are sent as directed.

- **emerg** – specifies emergency messages (**LOG_EMERG**). These messages are not distributed to all users. **LOG_EMERG** priority messages can be logged into a separate file for reviewing.
- **alert** – specifies important messages (**LOG_ALERT**), such as a serious hardware error. These messages are distributed to all users.
- **crit** – specifies critical messages not classified as errors (**LOG_CRIT**), such as improper login attempts. **LOG_CRIT** and higher-priority messages are sent to the system console.
- **err** – specifies messages that represent error conditions (**LOG_ERR**), such as an unsuccessful disk write.
- **warning** – specifies messages for abnormal, but recoverable, conditions (**LOG_WARNING**).
- **notice** – specifies important informational messages (**LOG_NOTICE**). Messages without a priority designation are mapped into this priority message.
- **info** – specifies informational messages (**LOG_INFO**). These messages can be discarded, but are useful in analysing the system.
- **debug** – specifies debugging messages (**LOG_DEBUG**). These messages may be discarded.
- **none** – excludes the selected facility. This priority level is useful only if preceded by an entry with an ***** (asterisk) in the same selector field.

Destinations

Use the following message destinations in the action field:

- File name – full path name of a file opened in append mode

- @Host – host name, preceded by @ (at sign)
- User[**,** User][...] – user names
- ***** – all users.

Rotation

Use the following rotation keywords in the rotation field.

- **rotate** – this keyword must be specified after the action field.

- **size** – this keyword specifies that rotation is based on size. It is followed by a number and either a k (kilobytes) or m (megabytes).
- **time** – this keyword specifies that rotation is based on time. It is followed by a number and an h (hour), d (day), w (week), m (month), or y (year).
- **files** – this keyword specifies the total number of rotated files. It is followed by a number. If not specified, then there is an unlimited number of rotated files.
- **compress** – this keyword specifies that the saved rotated files will be compressed.
- **archive** – this keyword specifies that the saved rotated files will be copied to a directory. It is followed by the directory name.

The following are some examples of possible configuration rules for typical */etc/syslog.conf* file:

```
# mail messages, at debug or higher, go to Log file.
# File must exist; it will be rotated when it gets larger
# then 500 kilobytes or if a week passes,
#number of rotated files is limited to 10, use compression
# and store the archived files in /syslogfiles directory
mail.debug /usr/spool/mqueue/syslog rotate size 500k time 1w files 10
compress archive /syslogfiles
# all facilities, at debug and higher, go to console
```

```
*.debug /dev/console
# all facilities, at crit or higher, go to all users
# and sent to central logging host
*.crit *,@loghost
```

SYSLOG TIPS AND TRICKS

The following table details some of the programs that use syslog, including the facilities and levels they log to and a brief description of each program:

- **amd** – daemon – err-info – PD NFS automounter.
- **date** – auth – notice – sets time and date.
- **ftpd** – daemon – err-debug – FTP daemon.
- **halt/reboot** – auth – crit – shutdown programs.
- **inetd** – daemon – err, warning – Internet super-daemon.
- **login/rlogin** – auth – crit/info – login programs.
- **named** – daemon – err-info – name server daemon.
- **nnrpd** – news – crit-notice – PD newsreader.
- **ntpd** – daemon, user – crit-info – network time daemon.
- **passwd** – auth – err – password setting program.
- **popper** – local0 – notice, debug – PD Mac/PC mail system.

sendmail – mail – alert-debug – mail transport system.

su – auth – crite, notice – switch UID.

sudo – local2 – alert, notice – PD limited **su** program.

syslogd – syslog, mark – err-info – internal errors, timestamps.

tcpd – local7 – err-debug – PD TCP wrapper for **inetd**.

cron – cron,daemon – info – system task-scheduling daemon.

/unix – kern – varies – the kernel.

The following handy script can be used to debug your */etc/*

syslog.conf file. Please note that **cron** and **mark** facilities cannot be tested using logger.

```
#!/bin/csh
#
# Script to generate one log message per
# priority level per facility
#
foreach i  (auth authpriv daemon kern lpr mail news syslog user uucp
local0 local1 local2 local3 local4 local5 local6 local7)
  foreach k  (debug info notice warning err crit alert emerg)
    logger  -t TAG -i -p $i.$k "Test message facility $i priority $k"
  end
end
```

On AIX, syslogd is configured as a subsystem process, which belongs to a subsystem group called **ras**. It is configured to be executed from the */etc/rc.tcpip* script started by init from */etc/inittab*. The AIX version of syslogd has a fixed constant, NLOGS, which determines the total number of actions that can be specified in the configuration file. According to my checks, this number is set to 20. If you exceed this number, syslogd ignores the extra actions and quietly throws away log messages that would have been sent to them. When invoked with **–d** (debug) flag, syslogd displays a table of the facilities, levels, and actions specified in the */etc/syslogd.conf* file. Each incoming message is also echoed to the screen along with the information about how it is being processed. Here are a few rows from the output produced by **syslogd –d**:

```
X X 7 X X X X X X X X X X X X X X X X X X X X UNUSED:
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 X CONSOLE: /dev/console
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 X WALL:
X X X 5 X X X X X X X X X X X X X X X X X X X X CONSOLE: /dev/console
X X X 5 X X X X X X X X X X X X X X X X X X X X FILE: /nsr/logs/messages
X X X 5 X X X X X X X X X X X X X X X X X X X X USERS: operator,
X X X X X X X X X X X X X X 5 X X X X X X X X FILE: /nsr/logs/summary
X X X X X X X X X X X X X X 1 X X X X X X X X USERS: root, operator,
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 X FILE: /var/adm/ras/syslog
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 UNUSED:
```

In this table, columns correspond to facilities, and rows to actions. The values displayed are priority levels as represented internally; an X means none.

The lines listed with the tag UNUSED in this table have no actions associated with them – you still have fewer than NLOGS destinations. If you don't see any such lines, make sure that you have not exceeded the limit; check your configuration file by probing the last entry with a **logger** command. If you have forgotten to create the log file you should expect messages similar to the following:

```
cfline( mail.debug     /usr/spool/mqueue/syslog)
syslogd: /usr/spool/mqueue/syslog: errno = 2
logmsg: pri 53, flags 8, from hostname, msg syslogd: /usr/spool/mqueue/
syslog: errno = 2
```

syslog-ng is a **syslogd** replacement that adds new functionality, such as:

- Ability to filter messages based on message contents using regular expressions.
- More powerful and intuitive configuration scheme.
- More reliable log forwarding using TCP.
- Ability to remember all forwarding hops, which makes it ideal for firewalled environments.

In order to examine your logs effectively you can utilize **swatch** (the Simple WATCHer). Written 100% in Perl, **swatch** monitors logs as they're being written to and takes action when it finds something you've told it to look for. This simple, flexible, and useful tool is a must-have for any healthily fearful system administrator.

LogSentry (formerly **Logcheck**) automatically monitors your system logs and mails security violations to you on a periodic basis. It is based on a program that ships with the TIS Gauntlet firewall, but has been improved in many ways to make it work nicely for normal system auditing. The program is available under GNU licence.

If you plan to use the network redirection feature of syslogd, it is very important to hide the whole network behind the firewall with disabled access to syslogd port UDP 514 in order to prevent remote denial-of-service attacks.

REFERENCES

1 Aeleen Frisch. *Essential System Administration*, Third Edition, O'Reilly & Associates, Inc.

2 Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein. *Unix System Administration Handbook*, Third Edition, Prentice Hall PTR

3 http://www.balabit.hu/en/downloads/syslog-ng/ **syslog-ng** home page

4 http://www.stanford.edu/~atkins/swatch, **swatch** home page

5 http://www.psionic.com/products/logsentry.html, **LogSentry** home page.

Alex Polyak
System Engineer
APS (Israel)

Conditional operators

The shell provides two special command terminators, && and ||, which are used to separate commands and control the way in which they are executed. The && operator has nothing to do with the & command terminator, and the || operator has nothing to do with the pipe character |.

Most things that you can accomplish with these operators can also be done using the **if** command. They provide concise alternatives to **if**, but for complex constructions inexperienced shell programmers often find them difficult to use and interpret; the logic is often easier to follow using multiple **if** statements.

THE && OPERATOR

You can use the && operator to separate two commands when you want the second command to be executed *only if* the first

command is successful. For example:

```
test $# -eq 1 && find / -name $1 -print
```

or:

```
[ $# -eq 1 ]   && find / -name $1 -print
```

or:

```
[[ $# -eq 1 ]] && find / -name $1 -print
```

The last construction is more commonly used and the line has the same effect as the following **if** command:

```
if [[ $# -eq 1 ]]
then
find / -name $1 -print
fi
```

You can construct longer lists of commands separated with && operators. Consider the following example of a command list:

```
command1 && command2 && . . . commandn
```

The commands are executed in a sequence from left to right, so long as the preceding command has an exit status of 0. If one of the commands returns a non-zero value, the remaining commands will not be executed and the return value of the entire list will be non-zero. If all the commands are executed, the return value will be that of the last command.

Here is an example of a list that contains three commands:

```
[[ $# -eq 2 ]] && [[ -f "$2" ]] && grep "$1" $2
```

In this example, the second command will be run only if there are two arguments, and the third command will be run only if the second argument is a valid filename.

Create a shell script that contains just the line shown above. Test what happens in the following cases, and each time view the exit status from your script:

1 Run the script without any arguments, and then with one argument, and note what happens.

2 Invoke the script again with the second argument being the

name of a file in your current directory. The first argument should be a string that is contained within the file.

3 Try it again, with a second argument that is not the name of a file and note what happens.

You can make the execution of a sequence of commands depend upon the exit status of a preceding command by enclosing the sequence of commands in curly braces to create blocks of code. For example:

```
command1 && { command2; command3; ... commandn; }
```

In a command list of this form, the commands that are enclosed in curly braces will be run if, and only if, the first command is successful.

If the curly braces were not used, the execution of the second command would depend on the return value of the first, but the remaining commands, *command3* to *commandn*, would be run regardless of what the first returned.

You should remember the following points about the use of curly braces:

- The braces must be separated by spaces from adjacent characters.
- Each command inside the braces, including the last command, must be followed by a command separator, which, in the above example, is the semi-colon. You could, of course, put the commands on separate lines so that the command separators then become carriage returns.

You can try the following command by creating a simple shell script that contains:

```
[[ ! -f $1 ]] && { print $1 does not exist ; exit 1; }
print $1 exists
```

Observe the difference in the messages when you run the script with a command line argument which is a filename that does not exist in your current directory, and with one which does exist.

The first command checks whether the specified file does not exist. If this is the case the test returns a zero exit status, and the command list enclosed within the curly braces is then run; after printing the message the script exits with a non-zero exit status. If the file does exist, the script continues on the next line, prints a different message, and exits normally.

This method of grouping commands is very common in script writing, where it is frequently used to test a condition, and then print an error message and exit if the condition is not true.

Also, consider what would happen if a command similar to the following were used instead:

```
[[ ! -f $1 ]] && print $1 does not exist && exit 1
```

In this example you will achieve the same result when the file does not exist. However, exit 1 will be executed only if the preceding commands are successful, and while it is unlikely that **print** will return anything other than 0, in other situations you may not be using a command quite so simple, so that your script may not exit when you want it to.

THE || OPERATOR

You can think of the *&&* operator as performing a *logical and* operation, since the return value of the command list will be true, if, and only if, all of the commands in the list are true. As soon as one of the commands fails, the shell knows that the list is false and it does not run the remaining commands.

In a similar way, you can think of the **||** operator as performing a *logical or* operation. A sequence of commands separated by **||** operators is executed sequentially from left to right until one of the commands is successful (true).

As soon as one command is successful, the shell decides that the entire list is successful and does not run the remaining commands. If none of the commands are successful, the return value of the list will be non-zero.

Consider the following command:

```
[[ -f $1 ]] || print $1 does not exist
```

If the $1 file exists, the test will be true, and **print** will not be executed. If $1 does not exist, the test will return a non-zero exit status, and **print** will display its message.

Curly braces are also useful for grouping commands with this operator, so that our simple script testing for the existence of a file can be written as follows:

```
{ [ -f $1 ]] || { print $1 does not exist; exit 1; }
print $1 exists
```

You can see that to achieve the same result we now test whether the file does exist, whereas with && we used reverse logic and tested for its non-existence.

COMBINING TEST EXPRESSIONS

In the previous examples using conditional operators we have usually performed a single test, and then, depending on its results, executed one or more commands. Let us now consider how we can combine multiple tests, so that if one condition is true *and/or* a second (or more) condition is true, we continue with the logic to execute further commands.

There are two ways in which you can combine test expressions. The first uses the binary **and** (**-a**) and the binary **or** (**-o**) operators, and the syntax can only be used within the single square brackets test. For example:

```
[ "$i" -ge 4 -a "$i" -lt 10 ]
[ "$i" -lt 4 -o "$i" -ge 10 ]
```

The first statement tests that the variable *i* is greater than or equal to 4, and less than 10, so that *i* can take a value from 4 to 9 inclusive for the test to be true. The second statement tests whether *i* is less than 4, or greater than or equal to 10, so that in this case *i* cannot have a value from 4 to 9 inclusive for the test to be true.

The second method uses the conditional operators, && and **||**, so that the above tests can also be written:

```
test "$i" -ge 4 && test "$i" -lt 10
test "$i" -lt 4 || test "$i" -ge 10
```

or:

```
[ "$i" -ge 4 ] && [ "$i" -lt 10 ]
[ "$i" -lt 4 ] || [ "$i" -ge 10 ]
```

or:

```
[[ $i -ge 4 ]] && [[ $i -lt 10 ]]
[[ $i -lt 4 ]] || [[ $i -ge 10 ]]
```

It is also possible to group expressions by using parentheses. For example:

```
[ \( "$1" -lt 3 -o "$1" -gt 7 \) -a "$2" -eq 2 ]
```

In the above example, the system first tests that $1 is less than 3 or greater than 7, and then that $2 equals 2. If these conditions are all satisfied, it returns the value 0. Since parentheses are meaningful to the shell, they must be escaped for this particular usage.

To achieve the same result using conditional operators the syntax would look like the following:

```
{ [[ $1 -lt 3 ]] || [[ $1 -gt 7 ]] } && [[ $2 -eq 2 ]]
```

or:

```
([[ $1 -lt 3 ]]  || [[ $1 -gt 7 ]])  && [[ $2 -eq 2 ]]
```

or:

```
[[ $1 -lt 3 ]]   || [[ $1 -gt 7 ]]   && [[ $2 -eq 2 ]]
```

For this particular command we can enclose the first two tests inside braces, or parentheses, or we can do without either, but this may not be the case for every test using combined operators since it will be dependent upon what you are trying to achieve. You will note that for constructions like this we do not need an additional command terminator, such as a colon, after the last test and before the closing brace.

You can get quite complex constructions when you combine tests, and also when you try to execute commands which are dependent on the results of individual tests. You can try this by creating a script containing the following lines:

```
{ [[ $1 -lt 3 ]]   && print first arg less than 3; } ||
{ [[ $1 -gt 7 ]]  && print first arg greater than 7; } &&
{ [[ $2 -eq 2 ]] && print second arg equals 2; }
```

We have split what could be a single line of code into three separate lines for readability. Each time the shell sees **||** or &&, it knows that the command line is not yet complete and will look on the following line for continuation of the syntax. You will also note that we now need a command terminator at the end of each of the **print** statements before the closing brace.

Try executing this script with various values for $1 and $2 and note the output. You can also try removing some of the brace pairs to see what difference this makes to the output.

THE SHIFT COMMAND

The **shift** command is a built-in shell command that moves the entire command line arguments one position to the left. After a **shift** command has been executed, the new value of $1 is whatever $2 used to be, the new value of $2 is whatever $3 used to be, and so on. The **shift** command also updates the value of $*.

Try the following example, which shows what the **shift** command does:

```
$ vi arg
print The first three args:
print $1; print $2; print $3
print '$*: \c'
print $*

shift 2
print The new first three args:
print $1; print $2; print $3;
print 'The new $*: \c'
print $*
```

When you execute the command as follows, you will see the following output:

```
$ arg one two three four five
The first three args
one
two
three
$*: one two three four five

The new first three args
three
four
five
The new $*: three four five
```

If you provide a numeric argument to **shift**, as we have done in our example, it will shift that many times; without an argument it shifts just the once. Note that there is no reverse shift, however. Once an argument has been shifted out of $1, it is gone forever, never to be recovered.

The **shift** command is usually used only with relatively simple command line options, and when complicated constructions are required we would normally use **getopts**. You might, for instance, use **shift** in a situation where you have an unknown number of command line arguments, which can be in any order and any quantity.

As an example, let's assume that you have problems remembering the command line options for the **chfs** command and so you create your own script, called **modfs**, which uses arguments you can easily remember and which are then converted to **chfs** options within the script. To make the logic relatively simple, let's assume that our script is called with the following syntax:

```
modfs number_of_blocks -l log_name filesystem_name
```

We have decided that it does not matter which order our options are in, provided that we have at least one in addition to the filesystem name; but if an option starts with a '**-**', then it will also take an argument. Here we have just the **-l** option, but we could also add the **-t** for type, **-c** for check, and so on, and these additional options would require **shift** to be used within a loop

until we had exhausted all the arguments.

Our script may contain something similar to the following lines (we have excluded code that builds up the command line to be executed and any other checks that may have to be made):

```
if [[ $# -lt 2 ]] || [[ $# -gt 4 ]]; then
      f_usage
      exit 1;;
fi

case $1 in
-l)
      OPT1=$1
      ARG1=$2
      shift 2;;
[1-9]*)
      SIZE=$1
      shift;;
*)
      f_usage
      exit 1;;
esac

if [[ $# -eq 1 ]]; then
      FILESYSTEM=$1
      f_chk_valid_filesystem
elif [[ $# -eq 2 ]]; then
      if [[ $1 != [1-9]* ]]; then
          f_usage
          exit 1
      fi
      SIZE=$1
      FILESYSTEM=$2
      f_chk_valid_filesystem
elif [[ $# -eq 3 ]]; then
      if [[ $1 != "-l" ]]; then
          f_usage
          exit 1
      fi
      OPT1=$1
      ARG1=$2
      FILESYSTEM=$3
      f_chk_valid_filesystem
else
      f_usage
      exit 1
fi
```

As you can see, the logic to check all the possible combinations for just a few command line arguments can become quite complex and using **shift** in this way will probably not be as efficient as using **getopts**, although for **getopts** to work we would have to introduce another option, **-s** say, to which our *number_of_blocks* value would have to be an argument.

Tonto Kowalski
Guru (UAE)

Make

INTRODUCTION

Make is a Unix utility that executes a list of shell commands associated with each target (could be an executable, an ASCII file, an object file, etc) typically to create or update a file of the same name. Makefile contains entries that describe how to bring a target up-to-date with respect to those on which it depends, which are called dependencies. Since each dependency is a target, it may have dependencies of its own. Targets, dependencies, and sub-dependencies comprise a tree structure that make traces when deciding whether or not to rebuild a target.

For example, let's say that currently we use the following list of commands in a file called *make_program.sh*. To make an executable called program:

```
$ cc   -c main.c
$ cc   -c iodat.c
$ cc   -c dorun.c
$ as   -o lo.o lo.s
$ cc -o program main.o iodat.o dorun.o lo.o /usr/fred/lib/crtn.a
```

Notes:

1 When we change main.c, we run *make_program.sh* again to **make** a program. Now *make_program.sh* does not have any

intelligence to figure out that the only program needing re-compiling is main.c and not the others. Therefore, it will recompile all the source files. In many cases, this unnecessary re-compilation may take some of the valuable time allocated for the build phase of the project.

2 The make utility provides this missing intelligence and a lot more to make this build process very efficient.

MAKEFILE

Makefile is the description file containing the rules and commands that **make** executes.

The name makefile is the default and can be replaced by any other, in which case **make** must be run with **-f** option for the alternative file name to be specified.

For example:

```
1    program : main.o  iodat.o   dorun.o  lo.o  /usr/fred/lib/crtn.a
2    cc  -o  program  main.o iodat.o dorun.o  lo.o  /usr/fred/lib/crtn.a
3    main.o : main.c
4             cc  -c  main.c
5    iodat..o  :  iodat.c
6             cc  -c  iodat.c
7    dorun.o  : dorun.c
8             cc  -c  dorun.c
9    lo.o   :  lo.s
1Ø            as   –o  lo.o  lo.s
```

Notes:

1 The numbers in the left margin are not part of the actual description file.

2 This description file contains file entries. Each entry consists of a line containing a colon (the dependency line or rules line), and one or more command lines which begin with a tab.

3 On each dependency line, to the left of the colon is a target and to the right of the colon are the target's prerequisites. The tab-indented command lines, therefore, show how to build the targets out of their prerequisites.

TARGET AND PROGRAM DEPENDENCY

Program A is said to depend on program B if program B needs to be integrated into program A before program A can function properly. In this case, program A can be classified as a target and program B as its dependency.

FILE EXTENSION

File extensions are one-, two-, or three-letter suffixes that are added to file names with a period or full stop (.).

Examples of file extension are:

- c
- pc
- as
- dat
- sql
- obj.

SUFFIX RULES

Suffix rules are the pre-defined rules that govern the dependency processing for a target based exclusively on file extensions.

WRITING SUFFIX RULES

Standard **make** allows processing rules to be built based around suffixes, not other parts of the filename. A necessary step in generalizing your commands is to choose a suffix that you use consistently on all files of a given type:

```
.SUFFIXES  :  .o  .c    .s
c.o  :
     ${CC}  ${CFLAGS}  -c  $<
.s.o  :
      ${AS}   ${ASFLAGS}    -o  $@  $<
```

Notes:

1 Line 1 must contain the keyword .SUFFIXES and all the file extensions that you would like **make** to examine while processing dependencies.

2 The second and third lines contain dependency processing rules for files with extensions .c and .o. The second line contains the statement in the form < make .o files from .c files >; and the third line contains the command to do it, which must start with a tab.

3 The fourth and the fifth lines contain dependency processing rules for files with extensions .s and .o. The fourth line contains the statement in the form < make .o files from .s files >; and the fifth line contains the command to do it, which must start with a tab.

4 Because line 1 states only file extensions .o, .c, and .s, **make** will not consider files with any other extension while making a particular target.

5 ${CC}, ${CFLAGS}, ${AS}, and ${ASFLAGS} are called macros. Macros will be defined in the makefile before they are used anywhere in the file.

MACROS AND DEFINING MACROS IN MAKEFILE

The makefile or the description file entries with the form:

```
CC =  cc
```

are macro definitions. Subsequent reference to ${CC} or $(CC) is interpreted as cc.

The macros are defined as:

```
<Macro Name>  =   <Macro Definition>
```

Notes:

1 White spaces before and after the equals sign are stripped off.

2 Macros could be written as ${MACRO} or $(MACRO).

COMMENTS IN MAKEFILE

The # sign can be used to comment a line or lines.

PREDEFINED OR INTERNAL MACROS

$?

$? gives the list of prerequisites that have been changed more recently than the current target. It can be used only in normal description file entries, not suffix rules.

For example:

```
libsql :   a.o  b.o  c.o
      ar  r  $@  $?
.SUFFIXES: .c  .o
.c.o:
      cc  -c     $<
```

Note: libsql is built using three object files – a.o, b.o, and c.o – which in turn are built from a.c, b.c, and c.c. The target build command states that when executing **ar** command, include only those object files that have changed. First time around, all the object files will be included in the building of library, libsql. Subsequently, if the source file b.c has changed, **make** will re-build b.o and issue the following command:

```
    ar  r  libsql  b.o
```

$@

$@ gives the name of the current target, except in description file entries for making libraries, where it becomes the library name. It can be used in both normal description file entries and suffix rules.

For example:

```
program:  a.o   b.o
      cc  -o $@    a.o   b.o
```

Note: the target program depends on a.o and b.o. When making the target, $@ takes the value of the name of the target, which is a program.

$$@

$$@ gives the name of the current target. It can be used only to the right of the colon in dependency lines.

$<

$< gives the name of the current prerequisite that has been modified more recently than the current target. It can be used only in suffix rules and .DEFAULT entries.

For example:

```
#  Suffix   rules
.SUFFIXES   .c  .o  .pc
.c.o:
          cc  -c  $<
```

Notes: the rules state how to make an .o file from a .c file. The **cc** command must have a source file name and $< evaluates to that file name that qualifies to be compiled. If we replace $< with a specific source file name, the **cc** command will only compile that source file even though there could be more than one source file that qualifies for re-compilation.

$*

$* gives the name without the suffix of the current prerequisite that has been modified more recently than the current target. It can be used only in suffix rules.

For example:

```
SUFFIXES : .o  .c  .pc
#=====================================================================
# How to make .c from .pc
#=====================================================================
pc.c :
      $(PROC) $(PROFLAGS) iname=$*.pc
      rm $*.lis 2>/dev/null
```

Notes:

1 When making .c from .pc, the iname parameter takes the source file name that has a file extension of .pc, and, therefore, $*.pc can provide that.

2 The process of making .c from .pc generates a .lis file with the same name and, therefore, $*.lis can be used to remove that file if required.

$%

$% gives the name of the corresponding .o file when the current target is a library module. It can be used in both normal description file entries and suffix rules.

ERROR AND EXIT STATUS

When a command produces an error (ie a non-zero exit status) the whole **make** is aborted.

But if you put a hyphen before any command, **make** will continue even when that command produces an error.

Note: you can force **make** to keep going regardless of command errors by putting the special .IGNORE target in the makefile (or description file). Alternatively, you can invoke **make** with the **-i** option in order to ignore errors.

PROCESSING RULES FOR MAKE

Standard **make** allows processing rules to be built only based around suffixes, not to other parts of the filename. A necessary step in generalizing your commands is to choose a suffix that you use consistently on all files of a given type.

EXAMPLE 1 MAKEFILE

```
CC=cc
#=================================================================
#  -c = compile only  -I = starting location for search
#=================================================================
```

```
INC_DIR=.
CFLAGS=-c -I $(INC_DIR)
#=====================================================================
#  +z = permit the use of null pointer
#=====================================================================
LDFLAGS=+z
PROC=$(ORACLE_HOME)/bin/proc
#=====================================================================
# lines=yes will preserve the original line number in .pc file.
#=====================================================================
PCCFLAGS=ireclen=132 oreclen=132 select_error=no mode=ORACLE lines=yes \
         sqlcheck=full userid=afz/afz
#=====================================================================
# Minimum Number of Libraries must be linked for embedded SQL program
#=====================================================================
LIBORA=$(ORACLE_HOME)/rdbms/lib/libora.a
OSNTAB=$(ORACLE_HOME)/rdbms/lib/osntab.o
NETLIBS=$(OSNTAB) $(ORACLE_HOME)/rdbms/lib/libsqlnet.a
LIBSQL14=$(ORACLE_HOME)/rdbms/lib/libsql14.a
PROLDLIBS=$(LIBSQL14) $(NETLIBS) $(LIBORA)
#=====================================================================
# Additional  Libraries  for  User exits ( make sure no space after \ )
#=====================================================================
FORMSLIB=$(ORACLE_HOME)/forms30/lib/iaddrvc.o \
         $(ORACLE_HOME)/forms30/lib/ifmdmf.o  \
         $(ORACLE_HOME)/forms30/lib/ifplut.o  \
         $(ORACLE_HOME)/forms30/lib/libforms30c.a \
         $(ORACLE_HOME)/forms30/lib/libforms30.a  \
         $(ORACLE_HOME)/forms30/lib/libforms30p.a \
         $(ORACLE_HOME)/orakit/lib/liboktc.a      \
         $(ORACLE_HOME)/orakit/lib/libokt.a       \
         $(ORACLE_HOME)/rdbms/lib/libpls.a        \
         $(ORACLE_HOME)/forms30/lib/libforms30c.a \
         $(ORACLE_HOME)/forms30/lib/libforms30.a  \
         $(ORACLE_HOME)/rdbms/lib/liboci14c.a
#=====================================================================
# Utility objects
#=====================================================================
UOBJS=uti_common.o  \
      uti_genmsg.o  \
      uti_dsp_bat.o
#=====================================================================
# Batch  objects
#=====================================================================
BOBJS=bch_db_access.o
#=====================================================================
# DEPENDENCIES
# Dependencies list (object list ) for Targets
#=====================================================================
ACC_SUM_TRA=acc_sum_tra.o \
```

```
            acc_app_aer.o \
            $(UOBJS)
ACC_END_DAY=acc_end_day.o \
            $(UOBJS)
ACC_CLS_YR=acc_cls_yr.o \
           $(UOBJS)
IAD3ØX=ue_acc_app_aer.o \
       acc_app_aer.o    \
       iapxtb.o         \
       $(UOBJS)
#=====================================================================
#TARGET BUILD
# Build  Targets
# $@ evaluates to current target
#=====================================================================
acc_sum_tra: $(ACC_SUM_TRA)
      $(CC)  -o $@ $(ACC_SUM_TRA) $(PROLDLIBS)
acc_end_day: $(ACC_END_DAY)
      $(CC)  -o $@ $(ACC_END_DAY) $(PROLDLIBS)
acc_cls_yr: $(ACC_CLS_YR)
      $(CC)  -o $@ $(ACC_CLS_YR) $(PROLDLIBS)
iad3ØX: $(IAD3ØX)
      $(CC) -o $@  $(IAD3ØX) $(FORMSLIB) $(PROLDLIBS)
#=====================================================================
# Description of suffix rules
# Order of any of these rules is not important
# $* macro evaluates to the current file name without the extension
# which make is dealing with
#=====================================================================
.SUFFIXES : .o  .c  .pc
#=====================================================================
# How to make .c from .pc
#=====================================================================
pc.c :
      $(PROC) $(PROFLAGS) iname=$*.pc
      rm $*.lis 2>/dev/null
#=====================================================================
# How to make .o from .c
#=====================================================================
.c.o :
      $(CC) $(CFLAGS) $*.c
#=====================================================================
# How to make .o from .pc
# remove .lis and .c files
#=====================================================================
.pc.o :
      $(PROC) $(PCCFLAGS) iname=$*.pc
      rm $*.lis 2>/dev/null
      $(CC) $(CFLAGS) $*.c
      rm $*.c 2>/dev/null
```

Notes:

1 Suffix rules show that files with extension .c, .o, and .pc are to be considered.

2 While defining macros, \ (backslash) can be used to continue the definition on the next line.

3 For each suffix rule, any valid command can be included.

4 Macros can be defined to include dependency lists.

5 The makefile has the following four targets:

```
o acc_sum_tra
o acc_end_day
o acc_cls_yr
o iad30x
```

6 To make all the afore-mentioned targets we need the following commands:

```
make –f  makefile     acc_sum_tra
make –f  makefile     acc_end_day
make –f  makefile     acc_cls_yr
make –f  makefile     iad30x
```

7 There is nothing in this makefile that will enable us to **make** all the targets with one call to **make**.

DUMMY TARGET ALL

In the example above we have defined four targets and, therefore, in order to **make** all these targets we had to call **make** four times with different target names. In order to avoid calling **make** four times, each calling for an individual target, we can define a dummy target called ALL, with all four targets as its dependants, as follows:

```
ALL  :  acc_sum_tra   acc_end_day  acc_cls_yr   iad30x
```

Then issue the following command:

```
make  -f   makefile   ALL
```

This will make all four targets.

EXAMPLE 2 MAKEFILE

```
#=================================================================
#   File          : make32.mk
#   Description   : Make file to build loadavg.so and plibmas.so
#   Usage         : make -f make32.mk all
#                     will make loadavg.so and plibmas.so
#                   make -f make32.mk loadavg
#                     will make loadavg.so
#                   make -f make32.mk plibmas
#                     will make plibmas.so
#=================================================================
include $(ORACLE_HOME)/rdbms/lib/env_rdbms.mk
# directory that contain oratypes.h and other oci demo program header
# files
INCLUDE= -I$(ORACLE_HOME)/rdbms/public -I$(ORACLE_HOME)/plsql/public -
I$(ORACLE_HOME)/network/public
#
CONFIG = $(ORACLE_HOME)/rdbms/lib/config.o
CC=cc -I. -g -v -Xc -D__EXTENSIONS__
PROC=$(ORACLE_HOME)/bin/proc
PROCFLAGS=ireclen=132 oreclen=132 select_error=no  \
          sqlcheck=$(SQLCHECK) userid='$(DBNAME)' \
          ltype=none xref=no mode=ansi lines=yes \
          define=__STDC__ \
          define=__EXTENSIONS__
PCCINCLUDE= include=$(ORACLE_HOME)/precomp/public
PCCI=-I$(ORACLE_HOME)/precomp/public
PROLDLIBS= $(LLIBSQL) $(TTLIBS)
.SUFFIXES:
.SUFFIXES: .o .c .pc
.pc.o:
      -@echo "*** Rule to make $@ from $<"
      $(PROC) iname=$*.pc $(PROCFLAGS) > $*.ERRS 2>&1
    $(CC) -c $(STD_INCLUDE) $*.c $(MASLIBS) $(PROLDLIBS) >> $*.ERRS 2>&1
      -rm -f $*.c
      -mv $*.o $*32.o
.c.o:
      $(ECHODO) $(CC) -c $(KPIC_OPTION) $(INCLUDE) >> $*.ERRS 2>&1 $<
      -mv $*.o $*32.o
#  dependency  line  for   dummy  target  ALL
ALL:     loadavg plibmas
#   dependency  line  for   target  loadave
loadavg: loadavg.o
      $(LD) $(SHARED_LDFLAG) loadavg.so loadavg32.o
#  dependency  line  for   target  plibmas
plibmas: lib_lock_file.o lib_unlock_file.o qsort_string.o \
            write_from_plsql_to_logfile.o
      $(LD) $(SHARED_LDFLAG) plibmas.so lib_lock_file32.o \
            lib_unlock_file32.o qsort_string32.o \
            write_from_plsql_to_logfile32.o
```

Notes:

1 Suffix rules do not have to be the last things defined in the makefile.

2 Notice the usage of dummy target, ALL.

3 Makefile can contain the **include** command to include definitions from other makefiles.

The syntax is:

```
include     file
```

The word *include* must begin the line and must be followed by a space or tab

4 Makefile can contain any operating system commands with '-'.

5 The processing that the command **make -f makefile loadavg** will perform is as follows:

```
ESTABLISH the prerequisites for target loadavg
RECORD loadavg.o as prerequisite
IF target exists in current directory
THEN
    IF target modification date > prerequisite modification date
            THEN
                    DISPLAY "Target is up to date "
                    EXIT
             END IF
  END IF
  RECORD no dependency line exists for prerequisite loadavg.o
  CONSIDER suffix rule .pc.o
  IF loadavg.pc does not exist
  THEN
           DISPLAY "Don't know how to make target loadavg.o "
           EXIT
  END IF
  IF modification date of loadavg.pc > loadavg.o
  THEN
           BUILD loadavg.o from loadavg.pc
  END IF
  ISSUE the command to make the target, loadavg
```

EXAMPLE 3

Requirement

We want to write a makefile that will build an executable called main that depends on four source files:

- main.pc
- a.pc
- b.pc
- c.pc.

Any time any of the source files is changed, **make** should reflect that change in the executable.

Makefile looks like:

```
main :  main.o   a.o   b.o   c.o
      cc  -o   $@  main.o  a.o  b.o  c.o

.SUFFIXES:
.SUFFIXES: .o .pc
.pc.o:
      proc  iname=$<
      cc  -c   $*.c
```

Notes:

1 Notice how the statement ‘target main depends on four source file main.pc, a.pc, b.pc, and c.pc’ has been turned into a dependency statement. Consider the following dependency line:

```
main  :  main.pc   a.pc  b.pc   a.pc
     cc  -o  $@  main.o  a.o  b.o  c.o
```

In this case, **make** won’t recognize a change in any of the source files because for this to happen there has to be a suffix rule such as .<something>.pc. Therefore, the prerequisites in the dependency lines must be the files that **make** can make in order to enforce the changes in the source files.

2 If you make a change in one of the source files, let's say a.pc, **make** will re-make a.o from a.pc and rebuild the target main.

3 The line *.SUFFIXES:* deletes all currently recognized suffixes. Therefore, the two lines:

```
.SUFFIXES:
.SUFFIXES: .o .pc
```

replace the current suffixes with .o and .pc. It will not nullify the current set of suffixes by adding the extra line; **make** will not recognize any changes in any of the .pc files. You can try this by commenting out the top line and then changing one of the source files and running **make** to build main. It will simply issue the command **cc -o $@ main.o a.o b.o c.o** without making .o from the changed source file.

TARGET WITHOUT ANY PREREQUISITES

```
clean:
      /bin/rm  -f     *.lis
```

Notes:

1 **Make** does allow targets to be defined without any pre-requisites. However, the colon must be present.

2 The command, **$make clean**, will execute the command **/bin/rm –f *.lis** as long as there is not an actual file called 'clean' in the current directory. **Make** will execute the command because **make** treats every non-existent target as an out-of-date target.

RUNNING MAKE WITHOUT A SPECIFIC TARGET

In this case **make** will build the first target contained in the makefile.

Note: it is possible to provide several target names in a single invocation of **make** as follows:

```
make      a.o  b.o  c.o
```

In general, recursive **make** solves problems when you want to determine information dynamically during your build and pass that information on to other parts of the build.

For example:

```
# Example for building demo OCI programs:
# 1. All OCI demos (including extdemo2):
#    make -f demo_rdbms.mk demos
# 2. A single OCI demo:
#    make -f demo_rdbms.mk build EXE=demo OBJS="demo.o ..."
#    eg make -f demo_rdbms.mk build EXE=oci02 OBJS=oci02.o
# 3. A single OCI demo with static libraries:
#    make -f demo_rdbms.mk build_static EXE=demo OBJS="demo.o ..."
#    eg make -f demo_rdbms.mk build_static EXE=oci02 OBJS=oci02.o
# 4. To re-generate shared library:
#    make -f demo_rdbms.mk generate_sharedlib
# Example for building demo DIRECT PATH API programs:
# 1. All DIRECT PATH API demos:
#    make -f demo_rdbms.mk demos_dp
# 2. A single DIRECT PATH API demo:
#    make -f demo_rdbms.mk build_dp EXE=demo OBJS="demo.o ..."
#    eg make -f demo_rdbms.mk build_dp EXE=cdemodp_lip
#                                      OBJS=cdemodp_lip.o
# Example for building external procedures demo programs:
# 1. All external procedure demos:
# 2. A single external procedure demo whose 3GL routines do not use
#    the  "with context" argument:
#    make -f demo_rdbms.mk extproc_no_context SHARED_LIBNAME=libname
#                                             OBJS="demo.o ..."
#    eg make -f demo_rdbms.mk extproc_no_context
#          SHARED_LIBNAME=epdemo.so OBJS="epdemo1.o epdemo2.o"
# 3. A single external procedure demo where one or more 3GL routines
#     use the  "with context" argument:
#    make -f demo_rdbms.mk extproc_with_context
#                           SHARED_LIBNAME=libname OBJS="demo.o ..."
#    eg make -f demo_rdbms.mk extproc_with_context
#           SHARED_LIBNAME=epdemo.so OBJS="epdemo1.o epdemo2.o"
#    eg make -f demo_rdbms.mk extproc_with_context
#                         SHARED_LIBNAME=extdemo2.so OBJS="extdemo2.o"
#    eg or For EXTDEMO2 DEMO ONLY: make -f demo_rdbms.mk demos
# 4. To link C++ demos:
#    make -f demo_rdbms.mk c++demos
# NOTE: 1. ORACLE_HOME must be either:
#               . set in the user's environment
#               . passed in on the command line
#               . defined in a modified version of this makefile
#       2. If the target platform support shared libraries (eg
```



```
#           Solaris look in the platform specific documentation for
#           Information about environment variables that need to be
#          Properly defined (eg LD_LIBRARY_PATH in Solaris).
include $(ORACLE_HOME)/rdbms/lib/env_rdbms.mk
RDBMSLIB=$(ORACLE_HOME)/rdbms/lib/
LDFLAGS=-L$(LIBHOME) -L$(ORACLE_HOME)/rdbms/lib
LLIBPSO='cat $(ORACLE_HOME)/rdbms/lib/psoliblist'
FC=f77
COB=cob
COBFLAGS=-C IBMCOMP -x
COBGNTFLAGS=-C IBMCOMP -u
CPLPL=CC
# directory that contain oratypes.h and other oci demo program header
# files
INCLUDE= -I$(ORACLE_HOME)/rdbms/demo -I$(ORACLE_HOME)/rdbms/public -
I$(ORACLE_HOME)/plsql/public -I$(ORACLE_HOME)/network/public
CONFIG = $(ORACLE_HOME)/rdbms/lib/config.o
# module to be used for linking with non-deferred option
# flag for linking with non-deferred option (default is deferred
# mode)
NONDEFER=false
# libraries for linking oci programs
OCISHAREDLIBS=$(TTLIBS) $(LLIBTHREAD)
OCISTATICLIBS=$(STATICTTLIBS) $(LLIBTHREAD)
PSOLIBLIST=$(ORACLE_HOME)/rdbms/lib/psoliblist
CLEANPSO=rm -f $(PSOLIBLIST); $(GENPSOLIB)
DOLIB=$(ORACLE_HOME)/lib/liborcaccel.a
DUMSDOLIB=$(ORACLE_HOME)/lib/liborcaccel_stub.a
REALSDOLIB=/usr/lpp/orcaccel/liborcaccel.a
PROC=$(ORACLE_HOME)/bin/proc
PCCINCLUDE= include=$(ORACLE_HOME)/precomp/public
PCCI=-I$(ORACLE_HOME)/precomp/public
USERID=scott/tiger
PCCPLSFLAGS= $(PCCINCLUDE) ireclen=132 oreclen=132 sqlcheck=full \
ltype=none user=$(USERID)
LLIBSQL= -lsql
PROLDLIBS= $(LLIBSQL) $(TTLIBS)
DEMO_MAKEFILE = demo_rdbms.mk
DEMOS = cdemo1 cdemo2 cdemo3 cdemo4 cdemo5 cdemo81 cdemo82 \
        cdemobj cdemolb cdemodsc cdemocor cdemolb2 cdemolbs \
        cdemodr1 cdemodr2 cdemodr3 cdemodsa obndra \
        cdemoext cdemothr cdemofil cdemofor \
        oci02 oci03 oci04 oci05 oci06 oci07 oci08 oci09 oci10 \
        oci11 oci12 oci13 oci14 oci15 oci16 oci17 oci18 oci19 oci20 \
        oci21 oci22 oci23 oci24 oci25 readpipe cdemosyev \
      ociaqdemo00 ociaqdemo01 ociaqdemo02 cdemoucb
DEMOS_DP = cdemodp_lip
C++DEMOS = cdemo6
.SUFFIXES: .o .cob .for .c
demos: $(DEMOS) extdemo2
```

```
demos_dp: $(DEMOS_DP)
generate_sharedlib:
      $(SILENT)$(ECHO) "Building client shared library ..."
      $(SILENT)$(ECHO) "Calling script $$ORACLE_HOME/bin/genclntsh ..."
      $(GENCLNTSH)
$(SILENT)$(ECHO) "The library is $$ORACLE_HOME/lib/libclntsh.so... DONE"
MAKECPLPLDEMO= \
      @if [ "$(NONDEFER)" = "true" -o "$(NONDEFER)" = "TRUE" ] ; then \
      $(ECHODO) $(CPLPL) $(LDFLAGS) -o $(EXE) $? $(NDFOPT)
$(OCISHAREDLIBS); \
      else \
      $(ECHODO) $(CPLPL) $(LDFLAGS) -o $(EXE) $? $(OCISHAREDLIBS); \
      fi
$(DEMOS):
      $(MAKE) -f $(DEMO_MAKEFILE) build EXE=$@ OBJS=$@.o
$(DEMOS_DP): cdemodp.c cdemodpØ.h cdemodp.h
      $(MAKE) -f $(DEMO_MAKEFILE) build_dp EXE=$@ OBJS=$@.o
c++demos: $(C++DEMOS)
$(C++DEMOS):
      $(MAKE) -f $(DEMO_MAKEFILE) buildc++ EXE=$@ OBJS=$@.o
buildc++: $(OBJS)
      $(MAKECPLPLDEMO)
.cc.o:
      $(CPLPL) -c $(KPIC_OPTION) $(INCLUDE) $<
build: $(LIBCLNTSH) $(OBJS)
      $(ECHODO) $(CC) $(LDFLAGS) -o $(EXE) $(OBJS) $(OCISHAREDLIBS)
extdemo2:
      $(MAKE) -f $(DEMO_MAKEFILE) extproc_with_context
SHARED_LIBNAME=extdemo2.so OBJS="extdemo2.o"
.c.o:
      $(ECHODO) $(CC) -c $(KPIC_OPTION) $(INCLUDE) $<
build_dp: $(LIBCLNTSH) $(OBJS) cdemodp.o
         $(ECHODO) $(CC) $(LDFLAGS) -o $(EXE) cdemodp.o $(OBJS)
$(OCISHAREDLIBS)
build_static: $(OBJS)
      $(ECHODO) $(CC) $(LDFLAGS) -o $(EXE) $(OBJS) $(SSDBED) $(DEF_ON) \
      $(OCISTATICLIBS)
extproc_no_context: $(OBJS)
      $(LD) $(SHARED_LDFLAG) $(SHARED_LIBNAME) $(OBJS)
extproc_with_context: $(OBJS) $(LIBCLNTSH)
     $(LD) $(SHARED_LDFLAG) $(SHARED_LIBNAME) $(OBJS) $(OCISHAREDLIBS) \
      $(LIBEXTP)
# these two targets are the same as the ones above extproc_no_context
#  and extproc_with_context. They are preserved for backward
# compatibility.
extproc_nocallback: $(OBJS)
      $(LD) $(SHARED_LDFLAG) $(SHARED_LIBNAME) $(OBJS)
extproc_callback: $(OBJS) $(LIBCLNTSH)
     $(LD) $(SHARED_LDFLAG) $(SHARED_LIBNAME) $(OBJS) $(OCISHAREDLIBS) \
      $(LIBEXTP)
```

```
clean:
      $(RM) -f $(DEMOS) extdemo2 *.o *.so
```

Notes:

1 Consider the command **make -f demo_rdbms.mk demos**. This is supposed to make all the following targets, which are its pre-requisites (expansion of macro ${DEMOS):

```
cdemo1 cdemo2 cdemo3 cdemo4 cdemo5 cdemo81 cdemo82
cdemobj cdemolb cdemodsc cdemocor cdemolb2 cdemolbs
cdemodr1 cdemodr2 cdemodr3 cdemodsa obndra
cdemoext cdemothr cdemofil cdemofor
oci02 oci03 oci04 oci05 oci06 oci07 oci08 oci09 oci10
oci11 oci12 oci13 oci14 oci15 oci16 oci17 oci18 oci19 oci20
oci21 oci22 oci23 oci24 oci25 readpipe cdemosyev
ociaqdemo00 ociaqdemo01 ociaqdemo02 cdemoucb
extdemo2
```

Once all the pre-requisites have been defined, we need to build a rule for each of them. This rule is defined using a recursive call to **make** as follows:

```
$(DEMOS):
$(MAKE) -f $(DEMO_MAKEFILE) build EXE=$@ OBJS=$@.o
```

The call to **make** would now look like the following:

```
/bin/make -f  demo_rdbms.mk build EXE=cdemo1 OBJS=cdemo1.o
/bin/make -f  demo_rdbms.mk build EXE=cdemo2 OBJS=cdemo2.o
```

and so on, where */bin/make* and *demo_rdbms.mk* are expanded from macros $(MAKE) and $(DEMO_MAKEFILE).

Without the design of recursive calls we would have had to hardcode all the build rules for each of the targets.

2 Target clean is defined to clear all unwanted files

RUN OPTIONS FOR MAKE

```
Make   [-f   makefile_name]  [options]  [targets]  [macro definitions]
```

Notes:

1 Options, targets, and macro definitions can be in any order.

2 Several options can be combined after a single hyphen.

3 The format of a macro definition is as:

```
Name=string
```

4 The most common options are:

- -e – let environment variables override macro definitions inside the makefile.
- - n – echo command lines, but do not execute them.
- - s – do not echo command lines.
- -t – touch target files (making them appear up to date) without executing any other commands.

A COMMON MISTAKE WHEN PREPARING A MAKEFILE

Missing out the tab before the command in a dependency line and in suffix rules is the most common mistake:

```
a :  b.o
      cc  -o $@  b.o

.SUFFIXES:  .c  .o
.c.o:
      cc  -c    $<
```

Note: both **cc** commands must start after a tab.

DEBUGGING MAKEFILE

If a makefile is not being run to completion, try running **make** with the **-d** option. As **make** goes through each dependency check and build targets, it displays descriptive messages.

For example:

```
$ make    -d   -f  makefile     mas_extract
MAKEFLAGS value:
   Building mas_extract.o using suffix rule for .pc.o because it is out
of date relative to mas_extract.pc
proc  iname=mas_extract
Pro*C/C++: Release 8.1.6.0.0 - Production on Sat Jun 22 15:12:01 2002
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
System default option values taken from: /u01/app/oracle/product/8.1.6/
precomp/a
```

```
dmin/pcscfg.cfg
cc  -xO2  -Xa  -xstrconst -xF  -mr  -xarch=v8 -xcache=16/32/1:1Ø24/64/1
-xchip=u
ltra -D_REENTRANT -K PIC -DPRECOMP  -I. -I/uØ1/app/oracle/product/8.1.6/
precomp/
public -I/uØ1/app/oracle/product/8.1.6/rdbms/public -I/uØ1/app/oracle/
product/8.
1.6/rdbms/demo -I/uØ1/app/oracle/product/8.1.6/plsql/public -I/uØ1/app/
oracle/product/8.1.6/network/public -DSLMXMX_ENABLE -DSLTS_ENABLE -
D_SVID_GETTOD    -c mas_extract.c
  Building mas_extract because it is out of date relative to
mas_extract.o
cc -o mas_extract  mas_extract.o  \
 -L/uØ1/app/oracle/product/8.1.6/lib/ -lclntsh
  'cat /uØ1/app/oracle/product/8.1.6/lib/sysliblist'  \
    -R/uØ1/app/oracle/product/8.1.6/lib -laio    -lm  -lthread
rm  mas_extract.o  mas_extract.c
```

Notes:

1 Consider the first message:

```
Building mas_extract.o using suffix rule for .pc.o because it is
out of date relative to mas_extract.pc
```

If the mas_extract.o file does not exist, **make** will always consider this as out -of-date relative to mas_extract.pc, otherwise **make** will check for the latest modification date on the mas_extract.o file. If the date is later than that of mas_extract.pc, it will not make that file.

2 Consider the second message:

```
Building mas_extract because it is out of date relative to
mas_extract.o
```

if mas_extract file does not exist, **make** will always consider this as out-of-date relative to mas_extract.o, otherwise **make** will check for the latest modification date on mas_extract file. If the date is later than that of mas_extract.o, it will not make that file. In this case, because mas_extract.o was made, **make** will certainly re-build the target mas_extract.

AIX news

IBM has announced Tivoli Identity Manager V2.1, which runs on AIX or Solaris, and centralizes the definition of users and provisioning of user services, provides role-based delegation of administrative privileges across organizational and geographical boundaries, provides users with self-care interfaces, automates the submission and approval of user administration requests and the implementation of administrative requests on the environment, and provides an application management toolkit.

IBM Tivoli Identity Manager V2.1 provides identity management by means of improved provisioning of identity data to more endpoints, cross-directory integration, and improved Web-based presentation and administration.

For further information contact your local IBM representative.
URL: http://www.tivoli.com/support/public/Prodman/public_manuals/td/ITIM/SC32-0827-00/en_US/HTML/id21prog.htm.

* * *

IBM has unveiled AIX 5L Version 5.2, which allows users to divide their server into smaller 'virtual' servers running either Unix or Linux. Like in the mainframe world, users can exploit the full power of their system by shifting workloads and changing resources transparently. If one partition experiences a problem and needs to be restarted it does not affect any of the other virtual servers on the system.

New to Version 5.2, IBM's implementation of dynamic Logical Partitioning (LPAR) allows system resources including processors, memory and, other components to be assigned to independent partitions, without rebooting the system. The ability to allocate resources without interruption eases system management and contributes to lower Total Cost of Ownership because the resources are better utilized.

Another feature of AIX 5L Version 5.2 is Capacity Upgrade on Demand (CUoD). Working synergistically, CUoD and dynamic LPAR help system administrators adapt to changing workloads and rapid growth without an interruption in service. Using CUoD and dynamic LPAR together, if an IBM eServer pSeries system has a failing processor, a new processor can be automatically brought online at no additional charge to the customer and with no interruption in service or performance degradation.

For further information contact your local IBM representative.
URL: http://www.ibm.com/servers/aix/os/52features.html.

* * *

IBM has announced WebSphere Voice Server Version 3.1 for Windows 2000 and AIX, which includes all the capability of WebSphere Voice Server V2.0, and provides the following new features: more natural sounding synthesized speech with a new Concatenative Text-To-Speech (TTS) engine, support for WebSphere Voice Response for AIX 3.1, and support for Web Sphere Voice Response for Windows 3.1.

For further information contact your local IBM representative.
URL: http://www.ibm.com.