107

AIX update

September 2004

In this issue

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

Phone: 214-340-5690
Fax: 214-341-7081

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Nicole Thomas
E-mail: nicole@xephon.com

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs $275.00 in the USA and Canada; £180.00 in the UK; £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 2000 issue, are available separately to subscribers for $24.00 (£16.00) 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.

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of $160 (£100 outside North America) per 1000 words and $80 (£50) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of $32 (£20) per 100 lines. 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.

Writing a daemon process

INTRODUCTION

Daemons are processes that live for a long time. Often they are started when the system is bootstrapped and terminated when the system is shut down. We say they run in the background, because they don't have any controlling terminal. Unix systems have numerous daemons that perform day-to-day activities.

Writing a daemon involves:

- Writing a daemon program.
- Creating a dedicated process in which to run the daemon program.

WRITING A DAEMON PROGRAM

The rules for writing a daemon program are as follows:

1 Construct a loop that will be executed at regular intervals in order to do its work.

2 Implement the regular intervals by issuing a sleep system call.

3 Implement proper signal handling, and designate a signal for termination.

4 Do not send any messages to stdout or stderr.

5 Implement an interface to a central error logging system.

FORK () AND EXEC ()

We need to be familiar with the fork () and exec () function calls, which will be used to create new processes and execute a desired program in that process.

fork ()

The only way a new process is created in Unix is by an existing process calling the fork function. Exceptions to this rule are three processes – swapper, init, and pagedaemon – which are created by the kernel as part of bootstrapping.

The new process created by fork is called a child process. This function is called once but returns twice. A process id of 0 is returned to the child process and the process id of the child process is returned to the parent process. Because it is not possible for a parent to obtain its child's process id, the fork () returns the child process id to its parent. The reason that the fork () returns 0 to the child process is that this can be used to establish whether or not this is the parent or child process. It is safe to return 0 by fork () to the child process because only the init process can have this process id and, therefore, when the child process id calls getpid (), it will receive its proper process id, and when it calls gettpid, it will receive its proper parent process id.

Both the child and parent continue executing with the instruction that follows the call to fork (). The child is a copy of the parent.

The following is a list of the properties of the parent that are inherited by the child:

- Real user id, real group id, effective user id, effective group id.
- Supplementary group ids.
- Process group id.
- Session id.
- Controlling terminal.
- Set-user-ID flag and set-group-ID flag.
- Current working directory.
- Root directory.

- File mode creation mask.
- Signal mask and dispositions.
- The close-on-exec flag for any open file descriptors.
- Environment.
- Attached shared memory segments.
- Resource limits.

The differences between the parent and child are:

- The return values from fork.
- The process ids are different.
- The two processes have different parent process ids – the parent process id of the child process is the parent; the parent process id of the parent does not change.
- The child's values for tms_utime, tms_stime, tms_cutime, and tms_ustime are set to 0.
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

exec ()

exec () is used to execute a new program in the child process that would be created by the fork (). When a process calls exec (), that process is completely replaced by the new program, and the new program starts executing as its main function. The process id does not change across an exec () because a new process is not created. Exec merely replaces the current process (its text, data, heap, and stack segments) with a brand new program from disk.

There are six different exec functions, and differences are mainly the different methods of passing various arguments.

DAEMON CHARACTERISTICS

Daemon characteristics are:

1 All the daemons run with superuser privilege (a user id of 0).

2 None of the daemons has a controlling terminal, and terminal name is set to a question mark when process details are displayed.

3 The parent of all daemons is the init process.

CODING RULES

The coding rules are:

1 The first thing to do is to call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group id of the parent, but gets a new process id – we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

2 Call setsid to create a new session, which leads to the following steps:

   – the process becomes a session leader of a new session.

   – the process becomes the process group leader of a new process group.

   – the process is tied up with no controlling terminal.

3 Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted filesystem. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted filesystem, that filesystem cannot be unmounted.

4 Set the file mode creation mask to 0. The file mode creation

mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.

5 Unneeded file descriptors should be closed. This prevents the daemon process from holding open any descriptors that it may have inherited from its parent (which could be shell or some other process). Exactly which descriptors to close, however, depends on the daemon.

6 Start executing the daemon program in the child process.

ERROR LOGGING

One problem a daemon has is to how to handle error messages. It can't just write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device since this can make administration difficult. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which file and to check those files on a regular basis. A central daemon error logging facility is required.

Use the following errlog () to log any errors to system log:

```
include <sys/err_rec.h>

int  errlog (  void  *ErrorStructure,  unsigned  int length )
```

For example:

```
char     err_msg[120];
char     module_details[] ="LogTrimmer" ;
int  buf_len;

struct  err_rec    ErrorStructure ;

ErrorStructure     *ptr ;
```

```
if   (  error  condition)
{

  ptr->error_id = <error_no> ;
  strcpy( ptr->resource_name, module_details) ;
  strcpy( ptr->detailed_data = err_msg) ;
  errlog (  ptr ,  size(ptr) ) ;


}
```

PROCESS CREATION AND EXECUTION

An example of process creation and the execution of a daemon program is shown below:

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <fctnl.h>
int   CreateDaemonProcess (void )
{
pid_t   pid;
/*
*  call fork ()
*/
if  (  pid  =  fork()  < Ø  )
/*
*  fork ()  failed
*/
    return (-1) ;
else   if ( pid  !=  Ø )
/*
* this  is  the  parent  process
*/
    exit(Ø)  ;
/*
*  child  or  daemon  process  continues
*/
setsid () ;
chdir ("/" );
umask(Ø);
/*
 *  execute  daemon   program  in  this  child  process
 */
    execle  ( "/home/admin/bin/LogTrimmer" );
return (Ø);
}
```

A common use for a daemon process is as a server process. In general, a server is a process that waits for a client to contact it, requesting some type of service. This communication between client and server process can be one-way or two-way. The area of interprocess communication is where numerous examples of two-way communication between a client and a server process can be found.

Arif Zaman
DBA/Developer (UK)

LPAR back-up over a network

We have several IBM pSeries Model 630s and 650s that replaced several SP/2 frames. The pSeries models don't have any tape drives, so we had to look for a way to back up the system data using mksysb. Because we shut down the SP/2 frames, we had no use for the Control Workstation that was attached to one of the SP/2 frames, so we decided to use the CWS machine after uninstalling all the PSSP stuff (we still call it CWS) and the tape drive attached to it. But then we ran into another problem. We could not back up all logical partitions (LPARs) at once, and on one tape. What we now do is back up all LPARs of one pSeries machine on one day, all LPARs of another pSeries machine the next day, and so on. Since we set up all LPARs using NIM we can restore the machine using NIM in the case of a disaster. A short description of how to use NIM commands to restore a machine is at the end of this article.

To achieve this we wrote two scripts, both started by an appropriate crontab entry. One script (later called net_backup.ksh) does the back-up on every LPAR and is located on all LPARs; the other script (later called tape_backup.ksh) writes the mksysb files to tape and is located on the Control Workstation.

Machine	*hostname*	*family-name*
pSeries 1:	u31aix	u3
	u32aix	u3
	u33aix	u3
pSeries 2:	u41aix	u4
	u42aix	u4
	u43aix	u4
	u44aix	u4
pSeries 3:	u51aix	u5
	u52aix	u5
	u53aix	u5
	u54aix	u5

Figure 1: The naming convention

We grouped all LPARs of one pSeries into ‘families’. We just use the hostnames because they fit into that schema. Because in our case the hostname is part of the name for the mksysb, it should be easy to adjust the filenames to fit every individual environment.

The naming convention for the machines is shown in Figure 1. This naming convention makes it easier later to back up all mksysb files. If your hostnames do not follow any conventions you can name the mksysb files with any name you like, so that a group of mksysb files are always written to tape at once. Just adjust the corresponding variables in the scripts. We can also group one ‘family’ in order to write all the mksysb file for all the machines in one ‘family’ to tape at once.

LPAR SCRIPT NET_BACKUP.KSH

We created a logical volume big enough to hold several mksysb files on the CWS, and mounted the filesystem */lpar/mksysb*. This filesystem holds the mksysb files of every single LPAR and the logs in a separate directory. This filesystem is mounted over NFS on every LPAR. The naming conventions are as follows:

- mksysb file:

  `/lpar/mksysb/<machinenname>.<date>`

  (eg u31aixot.21112003.)

- logfiles of mksysb:

  `/lpar/mksysb/net_backup/<date>.log`

- logfiles of tape back-up:

  `/lpar/mksysb/netbackup/tapebackup.log`

As already mentioned, the scripts are started via crontab. The back-ups for all LPARs on one pSeries machine are started on one day, for all the LPARs of the next pSeries on the next day, and so on (as Figure 2 shows). After all back-ups are finished, the mksysb files for all LPARs on one pSeries are written to tape. The time schedules represent my environment and must be adjusted to every individual environment.

The corresponding crontab entry looks like this:

```
ØØ 22 * * 1 /etc/net_backup.ksh
```

This means that the script runs every Monday at 22:00. As shown in Figure 2, this applies to all machines with a name starting u3*. On Tuesday, all machines starting with u4* will do their mksysb back-up, and so on. And here is the script:

```
#!/usr/bin/ksh
#
# Name: /etc/net_backup.ksh
#
# It will mount CWS:/lpar/mksysb to /mksysb
# logs are written to /mksysb/net_backup/$DATE.log
#
# Exit Code Ø = All Fine
# Exit Code 1 = Couldn't mount
# Exit Code 2 = Couldn't umount
#
# declare variables
BACKDIR=/mksysb
NAME=$(hostname)
DATE=$(date +%d%m%Y)
LOG=/mksysb/net_backup/$DATE.log
LOCALLOG=/tmp/$DATE.mksysb.log
```

Figure 2: Starting the scripts

Machine name	Script name	Day of back-up	Time of backup	Crontab Entry
u31aix	/etc/net_backup.ksh	Monday	20:00	00 20 * * 1 /etc/net_backup.ksh
u32aix	/etc/net_backup.ksh	Monday	20:15	15 20 * * 1 /etc/net_backup.ksh
u33aix	/etc/net_backup.ksh	Monday	20:30	30 20 * * 1 /etc/net_backup.ksh
CWS	/etc/tape_backup.ksh	Monday	21:30	30 23 * * 1 /etc/tape_backup 3
u41aix	/etc/net_backup.ksh	Tuesday	20:00	00 20 * * 2 /etc/net_backup.ksh
u42aix	/etc/net_backup.ksh	Tuesday	20:15	15 20 * * 2 /etc/net_backup.ksh
u43aix	/etc/net_backup.ksh	Tuesday	20:30	30 20 * * 2 /etc/net_backup.ksh
u44aix	/etc/net_backup.ksh	Tuesday	20:45	45 20 * * 2 /etc/net_backup.ksh
CWS	/etc/tape_backup.ksh	Tuesday	21:30	30 23 * * 2 /etc/tape_backup 4
u51aix	/etc/net_backup.ksh	Wednesday	20:00	00 20 * * 3 /etc/net_backup.ksh
u52aix	/etc/net_backup.ksh	Wednesday	20:15	15 20 * * 3 /etc/net_backup.ksh
u53aix	/etc/net_backup.ksh	Wednesday	20:30	30 20 * * 3 /etc/net_backup.ksh
u54aix	/etc/net_backup.ksh	Wednesday	20:45	45 20 * * 3 /etc/net_backup.ksh
CWS	/etc/tape_backup.ksh	Wednesday	21:30	30 23 * * 3 /etc/tape_backup 5

```
#
# after this, let's check whether mount works
# if it does, go ahead, otherwise, break
#
```

```
mount $BACKDIR
if [ $? -eq Ø ]
  then
  # We are fine just log that it works
  echo "$NAME.$DATE: Mount of $BACKDIR successful." >> $LOG
    else
  # No good. Lets log and stop right here
  echo "$NAME.f$DATE: Mount of $BACKDIR failed. Error. Exiting." >>
$LOCALLOG
  echo "$NAME.$DATE: No Backup of $NAME created" >> $LOCALLOG
  exit 1
fi

# Now we got a mount.
# The next step is creating our backup
echo "$NAME.$DATE: Starting mksysb at $(/usr/bin/date +%H:%M)" >> $LOG

/usr/bin/mksysb '-e' '-i' $BACKDIR/$NAME.$DATE $>> LOCALLOG

# Here we go. Lets see if backup was successful.
if [ $? -eq Ø ]
  then
  # Yes it was successful. Write a log message
  echo "$NAME.$DATE: Mksysb successful finished at $(/usr/bin/date
+%H:%M)" >> $LOG
  else
  echo "$NAME.$DATE: Mksysb finished with errors at $(/usr/bin/date
+%H:%M)." >> $LOG
  echo "$NAME.$DATE: See $LOCALLOG on $NAME for further informations."
>> $LOG
fi

umount $BACKDIR

# if this does not work, we alert the admin, to have a look at it.
# otherwise we are finished.
if [ $? -eq Ø ]
  then
      # We are done.
      exit Ø
  else
      # Write a log and go to sleep anywhere
      # This stuff is admin work not neccessary to care about
      echo "$NAME.DATE: umount of $BACKDIR unsuccessful. Please have a
look at                                                    $NAME" >>
$LOG
      exit 2
fi
```

```
# Should never get here.
exit Ø
```

CWS SCRIPT TAPE_BACKUP.KSH

This script mainly does two things. First it does a back-up of all mksysb files for all LPARs of one pSeries machine. After all mksysb files are successfully written to tape, it deletes all those files in order to have enough space for new back-ups. Don't forget to change the tapes after every back-up, label the tapes with the correct names, and store them in a safe place!

The corresponding crontab entry looks like this (see also Figure 2 above):

```
3Ø 23 * * 1 /etc/tape_backup 3
3Ø 23 * * 2 /etc/tape_backup 4
3Ø 23 * * 3 /etc/tape_backup 5
```

And here is the script:

```
#!/usr/bin/ksh
#
# At first, define variables before we go along
BACKDIR="/lpar/mksysb"
TAPE="/dev/rmtØ"
DATE=$(/usr/bin/date +%d%m%Y)
GROUP="u$1?aix"
LOG=$BACKDIR/net_backup/tapebackup.log

echo "$DATE: Starting tape backup at $(/usr/bin/date +%H:%M)" >> $LOG
/usr/bin/tar -cvf$TAPE $BACKDIR/$GROUP*.$DATE >> $LOG
if [ $? -eq Ø ]
  then
  # All went fine. Write log and go to next step
  echo "$DATE: $GROUP successfully backed up to tape. Backup ended $(/
usr/bin/date +%H:%M)" >> $LOG
  else
  # something went wrong. Write log and stop before deleting.
  echo "$DATE: $GROUP backup failed. Exiting." >> $LOG
  exit 1
fi

# Backup went fine
# Remove the old backup files
echo "$DATE: Deleting files of $GROUP from $DATE." >> $LOG
for i in $( ls -al $BACKDIR/$GROUP*.$DATE | awk '{ print $9 }' )
```

```
  do
  # echo what we delete and then delete it
  echo "$DATE: Deleting $i from filegroup $GROUP" >> $LOG
  rm $i >> $LOG
  if [ $? -eq 0 ]
    then
    # Deleted successful.
    echo "$DATE: Done." >> $LOG
    else
    # Something went wrong. Log information
    echo "$DATE: Deleting $i failed." >> $LOG
  fi
done

# Eject tape to prevent accidently overwriting it
echo "$DATE: Setting Tape to offline" >> $LOG
/usr/bin/tctl -f $TAPE offline >> $LOG
if [ $? -eq 0 ]
  then
  # Tape is ejected
  echo "$DATE: Done. Backup of $GROUP on $DATE finished" >> $LOG
  exit 0
  else
  # Tape is not ejected
  echo "$DATE: Failed. Eject tape manually from drive!" >> $LOG
  exit 2
fi

# Should never get here
exit 0
```

RESTORE SCENARIO

If you use NIM to set up all the machines, the restore scenario could look like the following.

Restore the tape with the proper mksysb files to the CWS, into the directory */lpar/mksysb*, using the following command:

```
tar -xvf/dev/rmt0 /lpar/mksysb
```

Now all the files for one pSeries are on disk on the CWS. You must copy the appropriate mksysb file to the NIM master for the machine to be restored. After that set up NIM to restore the machine using **smit** commands:

```
#smitty nim
```

```
–> Perform NIM Administration Tasks
–> Manage Resources
–> Define a Resource
o mksysb
* Resource Name:                 <name of mksysb>
* Serve for Resource:            <local NIM server>
* Location for Resource:         <where is the file on the NIM server>
<Enter>
```

After that, allocate the resources and set the NIM client machine to bos_inst using the appropriate NIM command and do a network boot.

Robert Schuster and Robert Frenzel
System Administrators (Germany)

Creating a cacheing DNS

Bind, the DNS package, can be downloaded from the Bull Web site (bind 9.2.1.0). Do not use the **bind** package shipped with the 5.2 distribution CDs because it contains an old package build. Even better is to download the source and build the package yourself from www.isc.org. Throughout this article I will use the names **bind** and **named** interchangeably because **bind** is synonymous with **named**.

A cacheing DNS does exactly what its name suggests, it caches your resolved queries. The process is as follows. A query is sent to the cacheing server to be resolved. It firsts looks in its cache to see whether the query has been resolved before. If it has, it will send the resolved query straight back to the client. If it does not have it in its cache, it will then go off to the next DNS (this part of the process is repeated, depending on whether the request is answered or not). The query will then be sent back to cacheing DNS and then sent back to the client. If the query is resolved, it will then reside in the cache's DNS. This cycle is repeated for each look-up.

At some point, a previously-resolved host may have changed

its IP address; so how can one know whether the local cache is up to date? Well, when a client tries to resolve a host that is not in the cache, the resolved query is sent back with a TTL (Time To Live) appended to it. This is the amount of time before the resolved query in the cache can be considered stale or suspect. So when a client tries to resolve a host again, the DNS will check its cache, see whether this host has previously been resolved, and if it has it will then check the TTL; if it has expired or been breached, the cacheing DNS will go off to the Internet and resolve it again, using the process mentioned earlier. One can also clear (delete) the DNS cache by restarting **named**.

WHY CHOOSE A CACHEING DNS?

In a nutshell, you choose a cacheing DNS to save time and reduce network traffic. If you have a ropey network, and your host times out a lot trying to query your master DNS, then install a cacheing server. If you are on a dial-up connection to the Internet, why bother trying to resolve hosts from your ISP's DNS all the time? Get the queries cached locally – install a cacheing DNS. Of course, by having a cacheing DNS, all the work is done by a main DNS and not your cacheing DNS.

SETTING UP A CACHEING DNS

Most if not all of the required configuration files will be installed with the **bind** package; all that is required is for the administrator to configure them. Sounds easy? It is. There are only three main files to configure for a basic cacheing server: *named.conf*, *localhost.rev*, and *root.cache*.

Assume for this article that our domain is called somecomany.co.uk

The first task is to decide the order of DNSs we wish to query. We point at ourselves with the loopback address of 127.0.0.1. The file which specifies the order of servers to be queried is */etc/resolv.conf*. To build a bit of redundancy into our queries, you should always specify at least two DNS servers. In the

example listing below, another DNS server is specified with an IP address of 192.168.4.50. In this file, one can also specify the default domain. This is the domain that will be appended to your queries when you try to resolve a host that does not have a dot in its name; in our case this would be the domain somecompany.co.uk, as in the example. Typically these would be internal hosts to the network. If your company has more than one internal domain, use the **search** statement instead of the domain statement in *resolv.conf*. For example if a company had two domains, called uk.somecompany.local and eu.somecompany.local, then one would use:

```
search uk.somecompany.local    eu.somecompany.local
```

In the following example using **nslookup**, we resolve an internal host, webserver1. Because the hostname does not contain any dots, the domain (somecompany.co.uk) is appended to it. Please try to use **dig** to resolve queries; **nslookup** is being gradually used less. It is used here for illustration purposes because of its minimal output.

```
Default Server:  localhost
Address:  127.0.0.1
> webserver1

Name:    webserver1.somecompany.co.uk
Address:  192.168.4.20
```

In the next example, we resolve a hostname that does contain dots in its name. The domain (somecompany.co.uk) does not get appended to it.

```
Default Server: localhost
Address: 127.0.0.1
> www.skynews.co.uk
Name:      www.skynews.co.uk
Address:   63.121.106.133
```

Example listing for */etc/resolv.conf*:

```
domain  somecompany.co.uk
nameserver      127.0.0.1
nameserver      192.168.4.50
```

A decision that also has to be taken is to decide on whether one

wants the local hosts resolved first by trying the */etc/hosts* file or using the DNS. If all queries should first try the */etc/hosts* and then go off to the DNS, edit the */etc/netsvc.conf* file and add the following:

```
order hosts, dns
```

Alternatively, if one wishes to first try the DNS then */etc/hosts*, use the following:

```
order dns, hosts
```

One can also use the environment variable NSORDER to specify the look-up order, like so;

```
# export NSORDER=hosts,bind
```

The decision one makes will be based on whether the AIX box has a loaded */etc/hosts* file. By that I mean many host entries.

The file called *named.conf*, located in */etc*, tells **named** what type of server it is to run, what type of options, the zone files to use, and the location of the named directory. A typical listing for */etc/named.conf* is shown below. The forward slashes '//' are comment lines.

```
// resolv.conf
// location of zone files
options {
            directory '/var/named';
      };

// root.cache file - hints file
zone "."  {
      type hint;
      file "root.cache"'
      };

// loopback file.
 Zone "0.0.127.in-addr.arpa"{
      Type master;
      File "localhost.rev"
      };
```

In the above listing, in the options clause, we are informing **named** where to find the zone files – I have specified */var/named*. The next line defines the zone and location of the

root.cache file (sometimes known as the hints file or *named.ca*). When **named** starts up, it goes off to locate the root servers that have been defined in root.cache. The "." means the root domain. The *type hint* tells **named** that there are pointers in this file pointing to the root servers that are defined within the file. Remember to copy the above zone files to the directory that is specified in the 'directory' options clause. Please note: a cacheing DNS does not contain any internal zones.

The loopback file *localhost.rev* (sometimes known as *named.local*), tells **named** that this is the master for its own loopback address. The zone "0.0.127.in-addr.arpa" is a special name denoting that this is a loopback assignment. Every DNS server must have a master, so what better place than the local loopback address? After all, the point of a cacheing server is to cut back on network traffic.

For security and access purposes, it is also a good idea to use access control lists (acl) – especially if the cacheing DNS is pointing to a main DNS on the Internet. One will undoubtedly want only internal clients to access the local cacheing DNS:

```
acl "internalhosts" { 192.168.2.0/24;  192.168.3.0/24; 192.168.4.0/24};
```

In the above command the acl list is given a name to reference the list by. In our example the list is called internalhosts, and only IP addresses that match this list will be allowed or denied to successfully query the DNS. The list contains in this example the (network address part) subnets 192.168.2.0 to 192.168.4.0. It is always a good idea to put the acl statement(s) at the top of */etc/named.conf*, because one cannot reference them before they are defined. Now within the options clause use:

```
allow-query {"internalhosts"};
```

We have now stated that IP addresses matching the list of internalhosts are allowed to query.

CACHE FOR INTERNAL OR EXTERNAL ROOT SERVERS

The root.cache file located in */var/named*, as detailed in */etc/*

named.conf, contains the names and the IP addresses of the root name servers on the Internet. When **named** is first started, this file is read to help **named** locate a root server. Once **named** has located one, its authoritative list of root servers are downloaded. The root.cache is installed by default when the named package is installed. You can also download the file from the Web: just do a search on 'root.cache file'; you will soon pick one up.

Alternatively, use **dig** to download the current root servers:

```
# dig @a.root-servers.net . ns > /var/named/root.cache
```

If the cacheing server is to query your main (master) internal DNS or your external ISP's DNS, just put their pointers in this file instead and leave it at that. So assuming our internal (master) DNS is called sample_main_dns and it has an IP of 192.168.4.50, we would have the following entry in */var/named/root.cache* for an internal root server:

```
.                   14D      IN      NS      sample_main_dns.
sample_main_dns.    14D              A       192.168.4.50
```

Please note the use of the dot (.) at the beginning of the host entry. The NS stands for nameserver; the A stands for addresses (as in IP address). The NS and A records are pointers to the root server. In 14D, the D stands for days, and means that these entries should be cached for 14 days before they are considered stale or untrustworthy. However, be aware that the time entries should be taken as is; in root.cache they are really there only for historical reasons. It is up to you to make sure the entries are up to date, so download the root.cache from the Internet at regular intervals – once a month should suffice. The format in the listing above must be used, however, whether for a downloaded Internet root.cache file, external or internal use, as just described.

LOOPBACK, THE LOCAL REVERSE LOOK-UP FILE

The local reverse look-up file is generated by default at installation time in most cases. The loopback address is

127.0.0.1. This is used for all loopback addresses, so do not change it! When **named** is started, it will query this address as the named server on your host. Of course, to point other hosts to this cacheing DNS, you must give it its machine IP address in the */etc/resolv.conf* and not the loopback address – but you knew that anyway, didn't you? The next listing shows a pretty much generic *localhost.rev* file for the loopback. Notice the first line, $TTL 1D. If another DNS queries our cacheing server, then the returned resolved query should be considered stale after one day. The @ means the current origin, which in the localhost.rev file means the localhost, though strictly speaking it is the 0.0.127.in-addr.arpa.

The loopback zone, in the *localhost.rev* file:

```
$TTL 1D
@           IN SOA   localhost. root.localhost. (
                     ØØ1     ; serial
                     3H      ; refresh
                     15M     ; retry
                     1W      ; expiry
                     1D )    ; minimum

                   IN       NS    localhost.
1                  IN       PTR   localhost.
```

The SOA (Start Of Authority) defines the settings for the zone. The *root.localhost* is the e-mail address for this zone. Most of the entries are here for historical reasons; they have no meaningful function. However, the PTR entry is used. It maps the loopback address to the localhost.

STARTING NAMED

To start **named** automatically when the AIX box comes up, make sure it is uncommented in */etc/rc.tcpip*:

```
start /usr/sbin/named "$src_running"
```

Alternatively, it can be started using:

```
# startsrc -s named
```

When configuring or altering **named**, you should also

continuously **tail** the messages file (in a new shell session) when re-starting named. The file */var/adm/messages* is where all **named**'s information is sent. For instance:

```
$ tail -f /var/adm/messages
```

To test the set-up, use either of the following resolving utilities, to make sure you are resolving hosts correctly:

```
nslookup, host or dig
```

Generally speaking, you can tell if a resolved host is in your cache, by using nslookup. If it comes back with 'Non-authoritative answer', this indicates that the host was resolved, but is not authoritative or it cannot guarantee the validity of the resolved query. You do not get this message if the DNS has to go out to a root server to get the answer.

That's it, you are ready to go – the basic DNS cacheing server is now configured.

BIND AND RNDC

The command line utility **rndc**, which comes with **bind**, allows one to remotely or locally administer **named**. Several commands are available; be sure to see the **man** page on this. However, to use **rndc**, one must first generate configuration files. The utility rndc-confgen, which comes bundled with **bind**, will print lines to standard output that must/should be added to *named.conf*. It also prints lines that it recommends should be used to create the main **rndc** configuration file, *rndc.conf*. The actual key file, *rndc.key*, is also created for you. This file is referenced by *rndc.conf* and *named.conf* respectively. Using these keys, **named** will accept connections only from a matching key over an authenticated channel, and this includes from the localhost. It is not necessary to have this feature, **named** will run OK without **rndc**, although from a security point of view it makes sound sense.

The following will generate an rndc-key (rndc.key) file for you, in */etc*:

```
# /usr/local/sbin/rndc-confgen -a
```

The contents of the created file are shown below:

```
key "rndc-key" {
       algorithm hmac-md5;
       secret "BivytJga3nHJ2GR4GLH00A==";
};
```

The actual key is called, surprisingly, rndc-key, and is a generated hashed md5.

To let rndc-confgen print lines informing the user of the recommended lines that should be put in the *named.conf* and *rndc.conf* file, simply run the utility with no options. Several lines are printed to standard output:

```
# /usr/local/sbin/rndc-confgen
# Start of rndc.conf
key "rndc-key" {
       algorithm hmac-md5;
       secret "BivytJga3nHJ2GR4GLH00A==";
};

options {
       default-key "rndc-key";
       default-server 127.0.0.1;
       default-port 953;
};
# End of rndc.conf

# Use with the following in named.conf,
#                                  adjusting the allow list as needed:
# key "rndc-key" {
#       algorithm hmac-md5;
#       secret "BivytJga3nHJ2GR4GLH00A==";
# };
#
# controls {
#       inet 127.0.0.1 port 953
#               allow { 127.0.0.1; } keys { "rndc-key"; };
# };
# End of named.conf
```

Copy the following contents from the output of rndc-confgen:

```
key "rndc-key" {
       algorithm hmac-md5;
       secret "BivytJga3nHJ2GR4GLH00A==";
};
```

```
options {
       default-key "rndc-key";
       default-server 127.0.0.1;
       default-port 953;
};
```

And paste the contents into a new file in */etc* called *rndc.conf*.

Copy the following contents from the output of rndc-confgen:

```
# key "rndc-key" {
       algorithm hmac-md5;
       secret "BivytJga3nHJ2GR4GLH00A==";
 };

 controls {
       inet 127.0.0.1 port 953
               allow { 127.0.0.1; } keys { "rndc-key"; };
 };
```

And append to the */etc/named.conf* file. Be sure to delete the comment lines (start with hashes). Notice that the port number used for the rndc channel communication is 953; normal **bind** queries use port 53.

The ‘rndc-key’ is the name referenced throughout the config files. Now **named** can be controlled from the localhost via the **rndc** command only if both keys match in each file. This is specified using the allow option ‘allow {127.0.0.1;}’. Please note, one can still start **named** from the command line as per normal, and stop **named** using the ‘kill’ command.

If you find you are having trouble with the **rndc** configuration, you can copy the above examples from this article onto their machine – it will work. Using **rndc** one can, amongst other tasks, reload the cache, gather statistics, or dump the cache. For example:

```
#/usr/local/sbin/rndc status
```

Bringing together */etc/named.conf* using the acl lists and rndc configuration, from what has been discussed in this article, the new look */etc/named.conf* is shown below:

```
// resolv.conf
// acl list, only local clients can query
```

```
acl "internalhosts" { 192.168.2.0/24;  192.168.3.0/24; 192.168.4.0/24};

// rndc control line for localhost
controls {
       inet 127.0.0.1 port 953
               allow { 127.0.0.1; } keys { "rndc-key"; };
 };

// the actual key !
 key "rndc-key" {
 algorithm      hmac-md5;
 secret "BivytJga3nHJ2GR4GLH00A==";
 };

// location of zone files
options {
            allow-query {"internalhosts"};
             directory '/var/named';
      };

// root.cache file - hints file
zone "."  {
      type hint;
      file "root.cache"'
      };

// loopback file.
 Zone "0.0.127.in-addr.arpa"{
      Type master;
      File "localhost.rev"
      };
```

Setting up a cacheing DNS is pretty straightforward, either with or without **rndc** support. I recommend clearing down the cache of a cacheing DNS at least once a week. This task can be run from cron.

David Tansley
Global Operations
ACE Overseas General (UK)

Teach me DB2 on AIX! – part 2

This month we continue our series of articles looking at DB2 UDB running on AIX and comparing it with DB2 on mainframes.

MORE UTILITIES

Only the following DB2 utilities will be contrasted: load utility, runstats utility, generate DDL, quiesce utility, and repair utility.

OS/390 (load utility):

- The OS/390 load utility loads one or more tables in a TS. It does not load the entire database as is the case in UDB. The tables to be loaded must exist. Any index defined on the table will be built automatically as part of the load. Load also checks Referential Integrity (RI).
- The OS/390 load has a new function called cross loader, which allows the OS/390 load to accept as input the contents of a cursor instead of a sequential dataset. The cursor can be reading a remote table. Here is an example of loading a mainframe table from the cursor input of an AIX UDB table. The cursor can be built just before the load statement using the three-part name of the remote UDB table.

```
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT *
FROM UDB.UDBØ3Ø.DEPARTMENT
ORDER BY 1 DESC
ENDEXEC

LOAD DATA
REPLACE
INCURSOR C1
INTO TABLE  UDBØ3Ø.DEPT
```

- To enable such load capability, the cross loader package DSNUGSQL has to be bound in DB2 OS/390 and in DB2

UDB. The TCP/IP connectivity and three-part name has to be set up in the CDB of DB2 OS/390.

- The cross loader function is supported from UDB to OS/390 and from UDB to UDB, but not from OS/390 to UDB.
- Keep in mind the difference between load REPLACE and load RESUME.
- Remember also that the dynamic features LISTDEF and TEMPLATES can be used effectively with OS/390 load.
- Think of the impact of loading tables that have triggers defined on them. These triggers in turn can induce sequences...!
- Think of the impact of loading tables that have columns defined as Identity column datatypes...!
- Remember that one cannot load Materialized Query Tables (MQT). These tables are refreshed and not loaded.
- The -Display utility(*) will show all the running OS/390 utilities and their status including the load utility. The equivalent of the Display Utility in UDB is the **Load Query** command, which shows the status of the UDB load while it is running.
- In OS/390 one can terminate any utility including the load utility by **-Term utility(<utility id>)**, whereas in UDB one has to reissue the same load statement syntax but replace the INSERT or REPLACE keywords with the keyword TERMINATE.

UDB (load utility):

- The traditional load utility in UDB is the IMPORT utility. But there is also an ‘official’ **LOAD** utility in UDB.
- The input files for the **IMPORT** utility as well as for the **LOAD** utility are files with the formats ASCII, delimited ASCII, or IXF.
- The target for the **IMPORT** utility as well as for the **LOAD**

utility can be only one table. Contrast that with OS/390, which can load several tables that are in a TS.

- The **LOAD** utility in UDB, as with OS/390, can build any index defined on the table during loading, but, unlike OS/390 load, it cannot check Referential Integrity (RI).
- The **IMPORT** utility in UDB, on the other hand, can build any index defined on the table and can also check RI. This is one important difference between the two UDB utilities (**LOAD** and **IMPORT**).
- The same option of REPLACE and RESUME of the OS/390 load utility exists in the UDB **LOAD** utility, but in UDB the RESUME keyword is replaced with a keyword INSERT.
- The UDB **LOAD** utility, just like the OS/390 load utility, expects the target DB2 table to exist.
- The UDB **IMPORT** utility also expects the target table to exist unless one includes the CREATE option in the **IMPORT** command. In that case the **IMPORT** utility will create the target table if it does not exist. That is also a difference between the two UDB load utilities.

  Here are some examples:

```
$DB2 "CONNECT TO nick"
$DB2 IMPORT FROM c:\filename.del OF DEL MESSAGES udb030.msg
        INSERT INTO udb030.tablename(columnname1,columnname2,etc);

$DB2 IMPORT FROM c:\filename.ixf OF IXF COMMITCOUNT 1000
       MESSAGES udb030.msg REPLACE_CREATE INTO udb030.tablename;

$DB2 LOAD FROM C:\filename.del OF DEL MESSAGES udb030.msg
         INSERT INTO udb030.tablename;

$DB2 LOAD FROM C:\filename.del OF DEL MESSAGES udb030.msg
         REPLACE INTO udb030.tablename;
```

- Is the creation of the DB2 table (if it did not exist) the only significant difference between UDB **LOAD** and UDB **IMPORT**? The answer is no. The real difference is performance. The UDB **LOAD** is a better performer because while the **IMPORT** utility loads one record at a time into the

DB2 table, the UDB **LOAD** takes several records from the input file and builds pages from them and loads several pages in the DB2 table at a time. This speeds things up.

- The **IMPORT** utility commits only after a successful import, otherwise it rolls back all the inserted records, ie either success of the entire load or failure of the entire load. There is nothing in between. So, one might ask, what happens if the **IMPORT** fails half way? My advice is to do what I do in OS/390. Before executing the **IMPORT** utility take a full image copy and a quiesce point. If the **IMPORT** utility fails then recover using the full image copy or recover to the taken quiesce point. This is the simplest strategy. I would personally use the same strategy if UDB **LOAD** fails too.
- How can one display the progress of the load utility in UDB? The **Load Query** command in UDB can show the status of the **LOAD**.

OS/390 (runstats utility):

- The main purpose of the runstats utility is to gather statistics on TSs, tables, columns, and indexes (in contrast to UDB where it gathers statistics on tables, columns, and indexes).
- The runstats utility records the statistical information in the OS/390 catalog so that the optimizer can choose an efficient access path to the data. An efficient access path reduces response time.
- One can interrogate the OS/390 catalog to see whether runstats has been run or not. A value of -1 in the relevant column indicates that runstats has not been run.
- One runs the runstats utility on TSs and their indexes, not on tables as in UDB.
- One runs the runstats utility frequently on TSs and indexes that are very volatile, so the catalog always contains up-to-date information about DB2 objects.
- One can run the runstats utility in SHERLEVEL REFERENCE or SHERLEVEL CHANGE.

UDB (runstats utility):

- The **runstats** command in UDB serves the same function as its counterpart in OS/390.
- Its main purpose is to gather statistics on tables, columns, and indexes (not on TSs as in OS/390) in order to allow the optimizer to choose an efficient access path, which in turn reduces response time.

  Here is an example:

```
$DB2 RUNSTATS ON TABLE udb030.tablename AND INDEXES ALL SHRLEVEL CHANGE
```

- The **runstats** command records the statistical information in the UDB SYSSTAT views of the UDB catalog, whereas in OS/390 the information was recorded in the SYSIBM tables.
- One can interrogate the SYSSTAT views to see whether **runstats** has been run or not. A value of -1 in the relevant column indicates that **runstats** has not been run.

  Here is an example of the interrogation:

```
SELECT * FROM SYSSTAT.TABLES;
```

- One runs **runstats** frequently on tables and indexes that are very volatile, so the catalog always contains up-to-date information about DB2 objects.
- One can run **runstats** in UDB in SHERLEVEL REFERENCE or SHERLEVEL CHANGE.

OS/390 (generate DDL):

- Sometimes the DBA needs to go to the DB2 catalog and generate DDL for an existing DB2 object or objects for the purpose of cloning. There are third-party products such as Bachman, BMC tools, CA tools, and Compuware tools, that can help the DBA to do DDL generation. There is no IBM utility as far as I know to generate mass DDL from the DB2 catalog. Some may say the IBM support utility DB2PLI8 is such a tool. I disagree. I personally think the DB2PLI8 is not suitable for mass DDL generation.

UDB (generate DDL):

- On the other hand, UDB has a very nice and effective utility that can generate DDL for an existing DB2 object or objects or for the entire UDB database for the purpose of cloning the database from one region to another. This utility is called **db2look**.

  Here is a sample of a **db2look** command to generate DDL for an existing database:

```
$db2 "CONNECT TO nickdatabase"
$db2look -d olddatabase  -a -o -l -e -x
outputddlforolddatabase.ddl -e -x -l
```

- Once the DDL is generated and stored in *outputddlforolddatabase* the DBA edits this output file and changes the name references of the old database to the new database name, and then executes the file by issuing the following command:

```
$db2 -tvf  outputddlforolddatabase.ddl.
```

- One can also use the **DESCRIBE** command to get the DDL for a table or the indexes on the table just like QMF in the OS/390. Here are two examples:

```
$DB2 "DESCRIBE TABLE <udb03.table name> SHOW DETAIL
$DB2 "DESCRIBE INDEXES FOR TABLE <udb03.table name> SHOW DETAIL
```

OS/390 (quiesce utility):

- Quiesce is used to establish a recovery point (an RBA or LRSN point) for a TS or TS SET, or indexspace or partition.
- This RBA or LRSN point will be recorded in the SYSIBM.SYSCOPY catalog table.
- The LISTDEF of DB2 OS/390 feature can be used neatly with the quiesce utility.

UDB (quiesce utility):

- The equivalent of the quiesce utility of OS/390 in UDB is the following command:

```
$DB2 QUIESCE TABLESPACES FOR TABLE <udb030.table name>
```

- This command will establish a recovery point (not RBA as in OS/390 but a timestamp). This recovery point is written to the HISTORY FILE, compared with SYSIBM.SYSCOPY of OS/390.

OS/390 (repair utility):

- Repair utility in OS/390 is used to repair data and remove pending statuses of TSs.

UDB (repair utility):

- There is no real equivalent to the repair utility of OS/390. The nearest equivalent is DB2DART, which can run only in read mode, unlike the repair of OS/390, where one can update and zap the data.

EXPLAIN FACILITIES

OS/390:

- The ultimate objective of the EXPLAIN functionality with OS/390 as well as UDB is to see the access path of a particular SQL statement chosen by the optimizer to access the data. The DBA then analyses the access decision made by the optimizer. As a result of the DBA analysis of the optimizer decision, the SQL statement may be modified or an index may be created on some column in the accessed table to achieve a good access path.
- To achieve the above objective, the DBA needs to create three EXPLAIN tables. The DDL for these three tables for OS/390 can be found in SDSNSAMP(dsntesc).
- The DBA can influence the optimizer decision by zapping various catalog tables with favourable statistics.
- Sometimes the optimizer's decision might not be the best access path despite all the available information. In this case the DBA needs to suggest to the optimizer a 'hint' that the optimizer can use to come up with a desired access

path. The hint can be inserted in the PLAN_TABLE, which is one of the three EXPLAIN tables in OS/390.

- There is an alternative method to interrogating the EXPLAIN tables manually via SQL. This alternative is to use a graphical tool called Visual Explain for OS/390. This is a nice graphical tool that should be installed on the DBA WS. It needs some kind of connectivity installed on the WS, such as DB2 Connect, to access the DB2 OS/390 machine.
- OS/390 has three EXPLAIN tables, UDB has seven. These tables are different from each other in structure and in the information they contain. Even the Visual Explain graphical tools are different from one platform to another. There is one Visual Explain product for OS/390 and one for UDB.
- One can invoke EXPLAIN also via the BIND command. This functionality is the same as in UDB.

UDB:

- The ultimate objective of the EXPLAIN functionality in UDB is the same as in OS/390, which is to see the access path of a particular SQL statement chosen by the optimizer to access the data.
- There are seven EXPLAIN tables that need to be created. To achieve that objective, the DBA needs to execute the DDL file supplied by IBM containing the definitions of these seven tables as follows:

```
$db2 -tvf $HOME/sqllib/misc/EXPLAIN.DDL
```

- Once the EXPLAIN tables are created, one can invoke the EXPLAIN function for a specific SQL statement. Here is an example of how to invoke the EXPLAIN from the command line interface:

```
EXPLAIN ALL WITH SNAPSHOT FOR "SELECT * FROM  udb030.nicktable"
```

- The DBA can interrogate the UDB optimizer decision manually via SQL or graphically using the Visual Explain product.

- The DBA can influence the optimizer by zapping various UDB catalog SYSSTAT views with favourable statistics. However, unlike the OS/390 DB2 DBA, you cannot give a hint to the optimizer for it to take it into consideration when calculating its access path. The hint functionality is not supported in UDB.
- There is another quick procedure to do EXPLAIN in UDB:
  - connect to your database.
  - create EXPLAIN tables as above.
  - just before your query, set a special UDB register to a value of EXPLAIN:

    ```
    $db2 "set current explain mode explain"
    ```

  - if the query is a static query then EXPLAIN the query by running the **db2expln** utility, which can be found in *$INSTHOME/sqllib/bin*.
  - if the query is a dynamic query then EXPLAIN the query by running the **dynexpln** utility, which can be found in *$INSTHOME/sqllib/bin*.
  - after doing the EXPLAIN, whether for static or dynamic SQL statements, run the **db2exfmt** utility, which goes to the explain tables and formats the result in an easy-to-read textual report. The **db2exfmt** can be found in *$INSTHOME/sqllib/bin*.
- So the whole quick procedure becomes:

```
$connect to SAMPLE
$db2 "set current explain mode explain"
      $db2 "select * from org"
      $db2exfmt
```

TRACES AND STATISTICS

OS/390:

- In OS/390 we have statistics or accounting reports for

performance (for UDB we have snapshot and events statistics).

- To get these performance reports one needs to start DB2 traces either via the ZPARM parameters or manually by issuing:

```
-start trace command
```

- The values in the ZPARM can be updated dynamically in Version 8 without needing to start or stop the DB2 subsystem (it is the same for UDB snapshot statistics, but not for event statistics).
- Once DB2 traces start, statistics start accumulating until one stops the trace.
- The destination of the trace statistics records is SMF, GTF, or Monitor.
- One can use DB2PM or equivalent third-party products to produce performance reports from the gathered statistical records.

UDB:

- For UDB there are two kinds of statistics that can be collected: one is called snapshot statistics (this may be roughly compared with the statistics coming from OS/390), the other is called events statistics.
  - Snapshot statistics:
    - Give a snapshot view of the state of the resource consumption at the instance level or at the application level from the time the DBA starts the trace.
    - Can be started in two ways:
      i by changing the dbm cfg parameters called switches (there are only six of them. Note that I said dbm cfg not db cfg). For example:

```
$db2 "UPDATE DBM CONFIGURATION USING <name of the dbm
```

```
parameter such as DFT_MON_STMT> ON
```

ii by explicitly updating one or more of the six snapshot switches using the following update command as an example:

```
$db2 "UPDATE MONITOR SWITCHES USING STATEMENT ON
```

(So do you see the similarity with OS/390 about starting the statistics via the ZPARM parameters or manually?)

- by the way, if the DBA wants to know all the available monitor switches s/he issues the following command:

```
$db2 "GET MONITOR SWITCHES"
```

- in both UDB and OS/390, once the DBA starts gathering the statistics, the process will continue until it is stopped.

- also in both UDB and OS/390, there is no need to recycle the DB2 subsystem or the instance for the statistics parameters to take effect.

- to view gathered snapshot statistics, for example for 'locks', one can issue the following UDB command:

```
$ db2 "GET SNAPSHOT FOR LOCKS ON Nickdatabase.
```

The result will be shown on the command line results window.

– Event statistics:

- the event statistics of UDB can be thought of as sleeping UDB objects for a particular database not for an instance.

- one has to create these DB2 objects using SQL DDL statements and, of course, like any other DB2 objects, the DBA has to give it a name.

- once these events objects are created, they are stored in SYSCAT.EVENTMONITORS and

SYSCAT.EVENTS tables of the UDB database catalog. These objects stay in the UDB catalog doing nothing. Whenever the DBA wants to take event statistics, the DBA needs to interrogate what kind of event objects he has created *a priori* in a particular database. Once the DBA knows what event objects are stored in the catalog, he can activate the desired ones.

- once activation occurs, the activated event monitor is not a sleeping object any more. It becomes active, collecting information as it was designed to do.
- here is an example of creating an event:

```
$DB2 "CREATE EVENT MONITOR <give it a name such
               as nickevntmonitor> FOR DEADLOCKS WRITE TO FILE
                    '/eventmonitors/deadlock/nickevntmonitor'"
```

- here is an example of how to activate an event:

```
$DB2 "SET EVENT MONITOR nickmonitor STATE =1"
```

- here is an example of how to deactivate an event:

```
$DB2 "SET EVENT MONITOR nickmonitor STATE =0"
```

- here is an example of how to view an event:

```
$DB2 "DB2EVMON -PATH '/eventmonitors/deadlock/nickevntmonitor'"
```

The output of the **db2evmon** utility will be displayed on screen by default, but one can direct it to a file for later analysis.

DB2 GOVERNOR

OS/390:

- Though the objective of the OS/390 governor and the UDB governor are the same, ie to monitor the excessive usage of database resources by applications (SQL statements) and to take corrective action based on supplied rules; the ways it is implemented in OS/390 and UDB are different.

- For example in OS/390 the governor places its rules in a specific table called DSNRLST*xx* situated in a specific TS called DSNRLS*xx* belonging to a specific database called DSNRLST. In UDB the governor is just a process (daemon) that monitors databases according to rules that are stored not in a table but in a file.
- In OS/390 the governor is called the Resource Limit Facility (RLF). One can start it through SPUFI with the command **-start RLIMIT**, or start it automatically via the ZPARMS when booting the DB2 subsystem.
- In OS/390 there is only one governor for the entire DB2 subsystem. However, there can be many tables containing the rules of engagement provided that only one table containing the rules is active at any one time.
- There are two kinds of governing in OS/390 – reactive governing, and predictive governing.

UDB:

- As was said above, the UDB governor is implemented differently from the OS/390 governor although the objective is the same.
- There can be several concurrent active UDB governors, each monitoring a different database and writing to its own log file. Contrast that with the OS/390 governor, which is one per DB2 subsystem.
- As far as I know there is no predictive governing in UDB as with OS/390.
- The UDB governor daemon is called db2gov.
- Here is one example of how to start the UDB governor:

```
$DB2GOV START nickdatabase
<name_of_cfg_file_that_contains_covernor_rules>
<name_of file_or_log_ that_contains_actions_taken_by_governor>
      $DB2GOV START nickdatabase nickrules.cfg nick.log
```

If the daemon governor was running for a long time, it could

potentially have created several log files such as nick.log.0, nick.log.1, or nick.log.2, etc. The DBA can list these log files and then use **DB2GOVLG** utility to read the chosen one. Here is an example of interrogating the UDB governor log file:

```
$DB2GOVLG nick.log.0
```

- Here is an example of how to stop the UDB governor:

```
$DB2GOV STOP nickdatabase
```

- There is no equivalent to a governor log file in OS/390. The MVS syslog and DSNMSTR address space in an OS/390 environment may contain some information in that regard.

BUFFER POOLS

OS/390:

- OS/390DB2 provides many options for data page sizes. The size of the data page is determined by the buffer pool in which one defines the tablespace. For example, a table space that is defined in a 4KB buffer pool has 4KB page sizes, and one that is defined in an 8KB buffer pool has 8KB page sizes. (Indexes – unlike TS – must always be defined in 4KB buffer pools.)
- The buffer pools are areas of memory within the DBM1 address space in which DB2 stores temporary pages of TSs and indexes. When an application needs a row from a DB2 table, DB2 retrieves the page containing the required row from DASD into the buffer pool area incurring an I/O. If the page is in the buffer pool area then no I/O is incurred. So buffer pools and their sizes are of the utmost importance to performance.
- In DB2 Version 7 and earlier, virtual memory was constrained because of DB2's 2GB limit on the size of the DB2 DBM1 address space (31-bit addressability). In its effort to satisfy more demands for memory usage for the buffers, DB2 Version 7 and before extended the buffer area above the

line to something called hyperpools and dataspace areas. But that was history. Now, with the z/OS and z/Series machines and the 64-bit addressability, the DBM1 address space has gone from 2GB to 16 exabytes (2**64). Thus larger buffer pools can be allocated in virtual memory and the definition of data spaces and hiperspaces previously designed to get around the old 2GB limit are no longer required.

- Buffer pools in OS/390 are allocated at installation or migration time. Page sizes for OS/390 buffer pools are 4KB, 8KB, 16KB, and 32KB.

  For each page size kind, there are the following predefined buffer pools:

  – 4KB page buffer pools are named BP0, BP1 to BP49.

  – 8KB page buffer pools are named BP8K0, BP8K1 to BP8K9.

  – 16KB page buffer pools are named BP16K0, BP16K1 to BP16K9.

  – 32KB page buffer pools are named BP32K, BP32K1 to BP32K9.

- One can change the attributes of OS/390 buffer pools and their sizes with the ALTER BUFFERPOOL command. The size is the number of pages in that particular pool.

- In OS/390 one assigns a tablespace or an index to a particular buffer pool by a clause in any of the following SQL statements: CREATE TABLESPACE, ALTER TABLESPACE, CREATE INDEX, or ALTER INDEX. The buffer has to exist before the DBA creates the TS. It's the same idea with UDB, the buffer has to exist before one creates the TS. The only difference between OS/390 and UDB in this regard is that the buffers in OS/390 are predefined at installation time whereas in UDB one explicitly creates a buffer.

- In OS/390 one can display active buffers via SPUFI or one can

interrogate the DB2 catalog to see which TS is associated with which buffer.

UDB:

- In DB2 UDB, unlike OS/390, the DBA can create bufferpools with the **CREATE BUFFERPOOL** command. Here are two examples:

```
CREATE BUFFERPOOL nickbpool
size 2000
pagesize 4096
```

Or:

```
CREATE BUFFERPOOL nickbpool
size 1000
pagesize 8192
```

- How many buffer pools can one create in UDB? The answer is 4096 pools.
- Despite the fact that UDB (like OS/390 DB2 Version 8) uses 64-bit addressability, the DB2 UDB still uses the equivalent concept of the old hiperpools of OS/390, which is called Extended Storage (estore memory) to provide a second level of cacheing for pages.
- There is a well-known diagram about memory usage that can be found in any book on UDB memory. Please find that diagram and see where the bufferpools and the extended memory storage (estore) reside.
- Remember, in UDB, the page size of the TS is entered as part of the **CREATE TABLESPACE** statement. Remember, again in UDB, a bufferpool with the correct page size needs to be created before creating the tablespace that uses this page size.
- In DB2 OS/390, there is no parameter in the CREATE TABLESPACE statement to indicate the page size to be used (as is the case in UDB). However, by specifying the predefined bufferpool to be used, the page size of the TS is set.

- The default buffer pool in UDB is called IBMDEFAULTBP and is 1000 4KB pages. It is created by the execution of the **CREATE DATABASE** command.
- Do you remember in OS/390 that there are read and write engines? They also exist in UDB, and are called I/O Servers. Their number is configurable by a parameter called num_ioservers in the db cfg file. These I/O Servers read pages from DASD into the buffer pools at the request of a db2agent process.
- There is one db2agent per application in UDB. However, in Version 8 with the advent of 'concentration pooling', a db2agent does not need to stay allocated until the application transaction finishes. It can serve other applications.
- To find out the current active bufferpools in UDB, one can issue the following two commands:

```
$DB2 "UPDATE MONITOR SWITCHES USING BUFFERPOOL ON"
$DB2 "GET SNAPSHOT FOR BUFFERPOLLS ON nickdatabase >
      /tmp/nickdatabase/bufferpools.txt
```

  Or one can interrogate the SYSCAT.BUFFERPOOLS catalog table.

EDM POOL

OS/390:

- In OS/390 there is a piece of memory called Environmental Descriptor Manager (EDM) pool. Among the many things it contains are package structures (ie access plans sections) and DBDs.

UDB:

- In UDB, the equivalent of the EDM pool is two pieces of memory or caches:
  - a cache for the packages, which is called PCKCACHESIZE.

– a cache to hold the DBDs known as catalog cache, but its real name is CATALOGCACHE_SZ

These parameters are updatable per database in the db.cfg file.

Nicola Nur
Senior DBA (Canada)

Parsing output of tapeutil command

For those of you with a tape library that supports the atape drivers, and hence provide a use of the tapeutil command, you may have found that tapeutil is not very script friendly. For example, if you want to write a back-up script that mounts tapes in the library via the robotics, it`s quite cumbersome. I use this script, parse_tu.sh, to put the output of tapeutil **–f /dev/smcx inventory** into a script-friendly format.

For example:

```
tapeutil -f /dev/smc0 inventory
```

would result in:

```
# tapeutil -f /dev/smc0 inventory
Reading element status...

Robot Address 1
  Robot State ..................... Normal
  ASC/ASCQ ........................ 0000
  Media Present ................... No
  Source Element Address Valid ... No
  Media Inverted .................. No
  Volume Tag .....................

Import/Export Station Address 16
  Import/Export State ............. Normal
  ASC/ASCQ ........................ 0000
  Media Present ................... No
  Import Enabled .................. Yes
```

```
  Export Enabled ................. Yes
  Robot Access Allowed ........... Yes
  Source Element Address Valid ... No
  Media Inverted ................. No
  Volume Tag .....................

Drive Address 256
  Drive State .................... Normal
  ASC/ASCQ ....................... 0000
  Media Present .................. Yes
  Robot Access Allowed ........... No
  Source Element Address ......... 4110
  Media Inverted ................. No
  Same Bus as Medium Changer ..... Yes
  SCSI Bus Address ............... 1
  Logical Unit Number Valid ...... No
  Volume Tag ..................... 045019L2

Drive Address 257
  Drive State .................... Normal
  ASC/ASCQ ....................... 0000
  Media Present .................. Yes
  Robot Access Allowed ........... No
  Source Element Address ......... 16
  Media Inverted ................. No
  Same Bus as Medium Changer ..... Yes
  SCSI Bus Address ............... 2
  Logical Unit Number Valid ...... No
  Volume Tag ..................... 045009L2

Slot Address 4096
  Slot State ..................... Normal
  ASC/ASCQ ....................... 0000
  Media Present .................. Yes
  Robot Access Allowed ........... Yes
  Source Element Address ......... 4096
  Media Inverted ................. No
  Volume Tag ..................... 045004L2

Slot Address 4097
  Slot State ..................... Normal
  ASC/ASCQ ....................... 0000
  Media Present .................. No
  Robot Access Allowed ........... Yes
  Source Element Address Valid ... No
  Media Inverted ................. No
  Volume Tag .....................
```

And so on...

By running

```
parse_tu.sh smcØ
```

I get:

```
# parse_tu.sh smcØ
Robot Address:  1  State:  Normal  Volume:
IO Address:  16  State:  Normal  Volume:
Drive Address:  256  State:  Normal  Volume:  Ø45Ø19L2
Drive Address:  257  State:  Normal  Volume:  Ø45ØØ9L2
Slot Address:  4Ø96  State:  Normal  Volume:  Ø45ØØ4L2
Slot Address:  4Ø97  State:  Normal  Volume:
Slot Address:  4Ø98  State:  Normal  Volume:  Ø45ØØ5L2
Slot Address:  4Ø99  State:  Normal  Volume:  Ø45ØØ7L2
Slot Address:  41ØØ  State:  Normal  Volume:  Ø45ØØ6L2
Slot Address:  41Ø1  State:  Normal  Volume:  Ø45ØØ8L2
Slot Address:  41Ø2  State:  Normal  Volume:  Ø45Ø1ØL2
Slot Address:  41Ø3  State:  Normal  Volume:  Ø45Ø13L2
Slot Address:  41Ø4  State:  Normal  Volume:  Ø45Ø14L2
```

Much easier to use in scripts!

Here is the script:

```
#!/usr/bin/ksh
>/tmp/tu_info_x1
tapeutil -f /dev/$1 inventory >/tmp/tu_info_x1
grep -E 'Robot Address|Robot State|Volume Tag|Import/Export Stat|Drive\
Address|Drive State|Slot Address|Slot State' /tmp/tu_info_x1\ >/tmp/
tu_info_x2;
#
#
while read one two three four; do
one_two='echo $one$two'
case $one_two in
  RobotAddress )
    export Robot_Address=$three
    read one two three four;
    export Robot_State=$four
    read one two three four;
    export Robot_Volume=$four;
    echo "Robot Address: " $Robot_Address " State: " $Robot_State "\
Volume: " $Robot_Volume;;
  Import/ExportStation )
    export IO_Address=$four;
    read one two three four;
    export IO_State=$four;
    read one two three four;
    export IO_Volume=$four;
```

```
    echo "IO Address: " $IO_Address " State: " $IO_State " Volume: "\
$IO_Volume;
;
  DriveAddress )
    export Drive_Address=$three;
    read one two three four;
    export Drive_State=$four;
    read one two three four;
    export Drive_Volume=$four;
    echo "Drive Address: " $Drive_Address " State: " $Drive_State "\
Volume: " $Drive_Volume;;
  SlotAddress )
    export Slot_Address=$three;
    read one two three four;
    export Slot_State=$four;
    read one two three four;
    export Slot_Volume=$four;
    echo "Slot Address: " $Slot_Address " State: " $Slot_State "\
Volume: " $Slot_Volume;;
esac
done </tmp/tu_info_x2
exit
#
```

David Miller
Database Architect
Baystate Health Systems (USA)

Software AG has announced Version 4.2 of Tamino XML Server, its native XML server. The new version provides accelerated data access, advanced security, and expanded query and text retrieval functions. It also offers XML-based message persistence for auditing and tracking, business document management, and a metadata repository in support of a Service-Oriented Architecture (SOA). These updates enable Tamino to better support multiple roles for enterprise integration and software developers.

Tamino XML Server version 4.2 is currently shipping in three editions – Enterprise Edition, Standard Edition, and Developer Edition. As well as AIX 5L V5.2 (64 bit), the product also runs on Linux for S/390 and zSeries, Windows XP Professional, Solaris 8 and 9 (64 bit), and HP-UX 11i (PA RISC 64).

For further information contact:
Software AG, 11190 Sunrise Valley Drive, Reston, VA 20191, USA.
Tel: (703) 860 5050.
URL: http://www2.softwareag.com/Corporate/News/latestnews/20040630_Tamino421_Release_page.asp.

* * *

OctetString has announced Version 3.0 of Virtual Directory Engine (VDE), its virtual technology for connecting and transforming identity information between enterprise systems software.

OctetString's VDE Suite connects applications to sources of identity information, including LDAP, RDBMS, Active Directory, or Windows NT Domain-based. Information from one or more of these identity repositories can be joined, federated, or otherwise virtually consolidated to present a single view of identities to applications via LDAP, XML, and JDBC. VDE is 100 percent Java and is certified on platforms that include AIX, HP-UX, Linux, Windows, and Solaris.

Version 3.0 delivers simplified configuration, accelerated deployment, and greater extensibility to users.

For further information contact:
Octet String, 10 N Martingale Road, 4th Floor, Schaumburg, IL 60173, USA.
Tel: (847) 358 9358.
URL: http://www.octetstring.com/products/VDE.php.

* * *

FileNet has announced Version 3.0 of P8, its ECM (Enterprise Content Management) platform. The new version now offers XML Web Service-based access to provide more platform-independent connectivity and interoperability, and supports requirements for Service Oriented Architectures (SOAs).

Using the product, businesses will be able to develop content-rich applications that deliver information when and where it is needed to empower business partners, suppliers, and employees to help decision-making.

FileNet P8 3.0 runs on AIX, Windows, Solaris, and HP/UX.

For further information contact:
FileNet, 3565 Harbor Blvd, Costa Mesa, CA 92626-1420, USA.
Tel: (714) 327 3400.
URL: http://www.filenet.com/English/News/Global-English/Current_Press_Releases/071904webserv.asp.