AIX update

35

September 1998

In this issue

AIX Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 550955
From USA: 01144 1635 33823
E-mail: HarryLewis@compuserve.com

North American office

Xephon/QNA
1301 West Highway 407, Suite 201-405
Lewisville, TX 75067
USA
Telephone: 940 455 7050

Contributions

If you have anything original to say about AIX, or any interesting experience to recount, why not spend an hour or two putting it on paper? The article need not be very long – two or three paragraphs could be sufficient. Not only will you actively be helping the free exchange of information, which benefits all AIX users, but you will also gain professional recognition for your expertise and that of your colleagues, as well as being paid a publication fee – Xephon pays at the rate of £170 ($250) per 1000 words for original material published in AIX Update.

To find out more about contributing an article, see *Notes for contributors* on Xephon's Web site, where you can download *Notes for contributors* in either text form or as an Adobe Acrobat file.

Editor

Harold Lewis

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £175.00 in the UK; $265.00 in the USA and Canada; £181.00 in Europe; £187.00 in Australasia and Japan; and £185.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1995 issue, are available separately to subscribers for £15.00 ($22.50) each including postage.

AIX Update on-line

Code from *AIX Update* is available from Xephon's Web page at www.xephon.com (you'll need the user-id shown on your address label to access it).

Thread concepts and programming

THE DEFINITION OF A THREAD

All modern Unix operating systems, and other ‘high-end’ operating systems, such as Windows NT, implement one of two methods of multiprocessing. The older method is to use multiple concurrent processes, while the new one is to use threads. A thread is defined as an independent flow of control that operates within the same address space as other independent flows of control within a process.

Traditionally, the creation of a new control flow in a Unix program requires the execution of the *fork()* system call. This system call is implemented by the kernel and results in the creation of a complete copy of the calling process, including all its private and system data, which demands substantial processor time and uses a lot of memory. The newly created process is completely separate from its parent and therefore has to use complicated inter-process communication primitives in order to coordinate activity and transfer data. By contrast, the creation of a new thread requires only two system calls and replication of the thread’s private data (about 64 KB), which therefore causes much less overhead.

Thread implementations are defined by following POSIX standards:

- POSIX 1003.4a Draft 4 (implemented in AIX V3 by the DCE Threads Library)
- POSIX 1003.1c Draft 7 (implemented in all variations of AIX V4)
- POSIX 1003.1c Draft 10 (implemented in AIX V4.3 in both 32-bit and 64-bit modes).

THREAD TYPES AND MODELS

There are three different types of thread: user, kernel, and kernel-only threads. User threads are created and manipulated by the user through functions defined in the *libpthreads.a* library. Their mapping to kernel threads is implementation-dependent. Kernel threads are operating system-managed entities that are handled by the system scheduler. They run in user mode when executing user functions and library calls, but switch to kernel mode when executing system calls invoked by the user. Kernel-only threads perform system-related tasks on behalf of kernel mode programs.

User threads are mapped to kernel threads by the threads library. The way in which this mapping is carried out is known as the ‘thread model’. There are three possible thread models available, which correspond to three different ways to map user threads to kernel threads: the ‘M:1’, ‘1:1’, and ‘M:N’ models.

The mapping of user threads to kernel threads is done using virtual processes. A virtual process (VP) is an implicit thread library entity that represents a CPU that’s able to run a thread. For a kernel thread, VPs represent real CPUs, while for the user threads they represent a kernel thread or a structure bound to a kernel thread.

The M:1 model

In the M:1 (*m-to-one*) model, all user threads are mapped to one kernel thread (or process), and all user threads run on one VP. The mapping and all thread programming features are handled by the thread library. This model can be used on any computer system including the traditional single-threaded systems. DCE threads were implemented in AIX V3 this way.

The 1:1 model

In the 1:1 (*one-to-one*) model, each user thread is mapped to one kernel thread and each thread runs on one VP. Most of the user thread’s programming is performed by the kernel thread. This model is implemented in AIX V4.1, AIX V4.2, and AIX V4.3.

The M:N model

In the M:N (*m-to-n*) model, user threads are mapped to a pool of kernel threads. A user thread can be bound to a specific VP or share a number of unbounded VPs. This is the most efficient and complex thread model; the user thread’s programming tasks are performed by thread libraries and kernel threads. This model is implemented in AIX V4.3.1.

THREAD SCHEDULING

Each thread object has a set of scheduling parameters. These parameters are associated with the thread when the thread is created by passing an appropriately initialized thread attribute to the *pthread_create()* function.

The scheduling priority of the thread is an integer value between 1 and 127, the highest priority being 127. The scheduling policy has three possible values: *SCHED_OTHER*, *SCHED_FIFO*, and *SHECD_RR*.

SCHED_OTHER is the default scheduling policy. It implements the standard AIX scheduling algorithm that decreases the priority of threads that are CPU intensive. *SCED_FIFO* is a strict 'first-in-first-out' algorithm that results in all threads with same priority running uninterrupted to completion. This policy should be used with extreme care as it can effectively destroy the performance benefits of multithreading. Another thread attribute that influences thread scheduling is the thread's *contention scope*.

THREAD CONTENTION SCOPE

The contention scope attribute can take one of two values: *PTHREAD_SCOPE_PROCESS* and *PTHREAD_SCOPE_SYSTEM*. 'Process' or 'local' contention scope specifies that the thread is to be scheduled in competition with all other local contention scope threads in the process. System or global contention scope specifies that the thread is to be scheduled in competition with all other threads in the system. The contention scope is only meaningful in a *M:N* library implementation. An attempt to set the contention scope attribute to *PTHREAD_SCOPE_PROCESS* on an operating system prior to AIX 4.3.1 will fail, producing an appropriate error message.

THREAD SMP SCHEDULING STRATEGY

When a thread is scheduled to run on a computer equipped with more than one CPU (SMP), a processor must be chosen to run the thread. There are two ways to perform this selection: selecting the same processor all or most of the time or selecting any available processor. The first method, known as ‘processor binding’, can reduce the need to access memory, as the processor’s cache may still contain data and instructions from the previous execution of the thread. The second method, called ‘opportunistic affinity’, tends to result in more efficient use of a computer’s processors. AIX V4 implements a strategy of opportunistic affinity, though it also allows threads to be bound to a specific processor by the execution of the *bindprocesor()* function of the *libpthread.a* library.

THREAD-ENABLED COMMANDS

With the introduction of kernel threads in AIX V4, a number of system monitoring commands were extended to report thread-related data.

ps - display process information

A new flag, **-m**, has been added to this command. When issued from the command line with this flag, **ps** displays information about threads (one line per thread). This flag should be combined with process selection flags (**e**, **a**, **k**, etc) and format flags (**I**, **F**, **o**, etc) to home in on the information about threads that the user requires. For instance, the command **ps -elm** produces long format display (**-l**) about all processes other than kernel processes (**-e**) and the threads associated with these processes (**-m**).

A new format flag, **-F**, has also been added to the command. Using this flag with a regular **ps** command, such as **ps -o THREAD**, causes additional thread information to be displayed.

netpmon – network I/O and network-related CPU statistics

This command has a new flag, **-t**, which prints CPU statistics on a per-thread basis. When this flag is used, each report line describing process statistics is followed by lines describing the CPU usage of each of the process’s threads.

Below is an example of part of the output of **netpmon -t**.

```
Process CPU Usage Statistics:

-----------------------------

                                                       Network

Process (top 20)              PID  CPU Time   CPU %   CPU %

-----------------------------------------------------------

netscape_aix4               11692   23.4825  67.442   0.000

    Thread id:              32077   23.4825  67.442   0.000

netscape_aix4               16732    6.3278  18.173   0.004

    Thread id:              34671    6.3278  18.173   0.004

X                            2358    2.1443   6.158   0.000

    Thread id:               3169    2.1443   6.158   0.000

netpmon                     16598    0.5460   1.568   0.000

    Thread id:              41877    0.5460   1.568   0.000

gil                          1032    0.2049   0.589   0.589

    Thread id:               2065    0.0534   0.153   0.153

    Thread id:               1807    0.0515   0.148   0.148

    Thread id:               1549    0.0477   0.137   0.137

    Thread id:               1291    0.0523   0.150   0.150

    Thread id:               1033    0.0000   0.000   0.000
```

tprof – reports CPU usage for the system and individual programs

The **-t** flag has been added to this command to constrain the report to a specific *process_id* and its children, adding a new *Thread Identification (TD)* column.

sar – reports system activity counters

While the format of the report hasn't changed, the meaning of the following flags is now subtly altered:

- **-q** refers to the average of threads and not the processes
- **-w** reports the number of thread, not process, switches per second.

vmstat – report statistics on kernel threads and other system activity

The *r*, *b*, and *cs* column headings respectively refer to:

- The number of kernel threads placed on the run queue per second
- The number of kernel threads placed on the wait queue per second
- The number of thread context switches placed on the run queue per second.

BASIC THREAD PROGRAMMING EXAMPLE

The following program displays the basic features of any thread-based program, including:

- Setting thread attributes to required values
- Creating threads
- Passing an argument to a thread
- Binding a thread to a processor
- Passing return data from the thread
- Terminating the thread.

Please note that this example (and any other programs that implement threads) should be compiled using either **xlc_r** or **xlC_r**. These commands assure compilation using the correct parameters and linkage with libraries that are thread-safe.

THREAD PROGRAM

```
#include <pthread.h>     /* For pthread-related macros and functions */
#include <stdio.h>       /* For formatted I/o */
#include <unistd.h>

void *Thread(void *arg)               /* Thread function */
{
  int rc;                             /* Return code */
  /* Bind thread to processor #0 */

  rc = bindprocessor(BINDTHREAD, thread_self(), 0);

  /* Report argument passed by the calling program */

  printf("String passed from the main program is: %s\n", (char *)arg);

  /* Terminate thread and pass argument back to main program */

  pthread_exit("Hear you main!");

}

void *main(int argc, char **argv)
{
  pthread_t thread;                 /* Thread variable */
  pthread_attr_t attr;              /* Thread attributes variable */
  char targ[] = "Hello thread!";    /* Argument to be passed to the
                                       thread */
  int rc;
  struct sched_param sched;         /* Scheduling attributes */
  char *thread_rc;                  /* Threads return value */

  /* Initialize thread attributes */

  pthread_attr_init(&attr);

  /* Set detachechstate attribute to CREATE_UNDETACHED to allow
     storage and successful retrieval of thread's return status */

  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);

  /* Set sceduling_policy and scheduling_priority attributes */

  sched.sched_policy = SCHED_RR;
  sched.sched_priority = 80;

  /* Create the thread and pass to it the argument */

  rc = pthread_create(&thread, &attr, Thread, (void *)targ);
  if (rc) {
    perror("Thread invocation failed!\n");
    exit(1);
  }

  /* Wait for threads termination and receive its return value */

  pthread_join(thread, (void *) &thread_rc);
  printf("Thread returned: %s\n", thread_rc);
  exit(0);
}
```

REFERENCES

1 *Introduction to Multithreaded Programming* by Chary Tamirisa, *AIXpert*, November 1994.

2 *Multithreaded Implementations and Comparisons, A White Paper*, Sun Microsystems, Part No: 96168-001 (1996).

3 *PThreads Primer* by Bill Lewis and Daniel Berg, SunSoft Press (1996).

4 *Programming with Threads* by Devang Shah, Steve Kleiman, and Bari Smaalders, SunSoft Press (1996).

5 *Thread Time* by Scott Norton and Mark Dipasquale, Prentice Hall (1996).

6 *Programming with POSIX Threads* by Dave Butenhof, Addison Wesley (1997).

A Polak
Systems Engineer
APS (Israel)

CPU usage monitor

CPU usage monitor is a shell script that plots a graph of CPU usage (in percentage terms) on the *y* coordinate against the number of required cycles (by issuing the command **sar 1 1** repeatedly) on *x* coordinate. The script runs under VT100 terminal emulation software for AIX. The actual CPU usage is worked out from the output of **sar 1 1** command, which takes one sample per second. The graph shows CPU usage in steps of 5% and should serve as a broad indicator of CPU usage.

LISTING

```
#########################################################################
#
#   Name        : cum  (CPU usage monitor)
#
#   Description : The script plots a graph of CPU usage against
#                 number of one second-interval cycles.
#
#                 It includes the following functions:
#
#                  o DefineVariables
#                  o DisplayMessage
#                  o MoveCursor
#                  o HandleInterrupt
#                  o GetRequiredNumberOfSamples
#                  o DrawCpuUsageGraoph
#                  o PopulateCpuUsageGraph
#
#   Notes       : 1 The command used to monitor CPU usage is sar 1 1,
#                   which takes one sample per second for all
#                   processors combined.
#
#                 2 The script repeats the sampling process and plots
#                   a graph showing the number of cycles input by the
#                   user.
#
#########################################################################

#########################################################################
#
#   Name           : DefineVariables
#
#   Description    : Defines all variables.
#
#########################################################################

DefineVariables  (  )
{

#
# define cursor home
#
X_HOME=1
Y_HOME=1
#
# define escape sequences
#
ESC="\0033["
RVON=_[7m                  # revrese video on
```

```
RVOFF=_[27m                 # reverse video off

BOLDON=_[1m                 # bold on
BOLDOFF=_[22m               # bold off

BON=_[5m                    # blinking on
BOFF=_[25m                  # blinking off

FEC=1                       # failure exit code
SEC=0                       # success exit code

SLEEP_DURATION=3            # number of seconds for sleep command
ERROR="${RVON}${BON}cum.sh:ERROR:${BOFF}"
INFO="${RVON}${BON}cum.sh:INFO:${BOFF}"

#
# define  signals
#
SIGNEXIT=0  ;  export SIGNEXIT     # normal exit
SIGHUP=1    ;  export SIGHUP       # session disconnected
SIGINT=2    ;  export SIGINT       # ctrl-c
SIGTERM=15  ;  export SIGTERM      # kill command
#
# define  message
#
INVALID_SAMPLE="\${NO_SAMPLES}, is out of range${RVOFF}"
INVALID_NUMBER="\${NO_SAMPLES}, is a bad number${RVOFF}"
INTERRUPT="Program Interrupted! Quitting early${RVOFF}"

}

####################################################################
#
#   Name           : MoveCursor
#
#   Description    : The function moves the cursor to a given point.
#
#   Input          : y coordinate value
#                    x coordinate value
#
####################################################################

MoveCursor (  )
{

YCOR=$1
XCOR=$2

print -n    "${ESC}${YCOR};${XCOR}H"
```

```
}

#####################################################################
#
#   Name           : DisplayMessage
#
#   Description    : The function displays a  given message
#
#  Input           : Message Type (E = Error, I = Information)
#                    Message Code
#
#####################################################################

DisplayMessage (  )
{

MESSAGE_TYPE=$1
MESSAGE_TEXT=`eval echo $2`

MoveCursor 24 1

if [ "${MESSAGE_TYPE}" = "E" ]
then
   echo  "`eval echo ${ERROR}`${MESSAGE_TEXT}\c"
else
   echo  "`eval echo ${INFO}`${MESSAGE_TEXT}\c"
fi

sleep ${SLEEP_DURATION}

}

#####################################################################
#
#  Name        : HandleInterrupt
#
#  Overview    : The function displays an appropriate message and
#                exits returning a failure code.
#
#####################################################################

HandleInterrupt ()
{

DisplayMessage I "${INTERRUPT}"

echo "${RVOFF}"
clear
# MoveCursor  ${Y_HOME} ${X_HOME}
```

```
exit  $FEC

}

######################################################################
#
#   Name         : DrawCpuUsageGraph
#
#   Description  : The function:
#
#                    o draws x and y coordinates
#                    o labels the coordinates
#                    o marks X coordinate with time interval
#                    o marks Y coordinate with perecent used
#
#   Notes        : 1 Cursor position (1, 1) is to top left hand
#                    corner of the screen.
#
#                  2 The y coordinate is marked from 0 to 100
#                    percent with inetrval of 10 percent.
#
#                  3 The x coordinate is marked from 0 to 50 with
#                    an interval of 5 cycles
#
######################################################################

DrawCpuUsageGraph  (  )
{

trap "HandleInterrupt " $SIGINT  $SIGTERM $SIGHUP

clear
#
# fix the coordinate for reference point (not necessarily the origin
# of the graph)
#
X_REF=10
Y_REF=3
#
# fix the coordinate for the first label (CPU usage) on the y
# coordinate
#
X_LABEL1=6
Y_LABEL1=1
#
# fix the coordinate for the second label ('^')
#
X_LABEL2=10
Y_LABEL2=2
#
```

```
# move the cursor to the right position for the first label
#
MoveCursor  ${Y_LABEL1} ${X_LABEL1}
#
# display the first label
#
echo "${BON}CPU  Usage(%)${BOFF}"
#
# move the cursor to the right position for the second label
#
MoveCursor  ${Y_LABEL2} ${X_LABEL2}
#
# display the second label
#
print -n  "^"
#
# move the cursor to the reference point
#
MoveCursor  ${Y_REF} ${X_REF}
#
# draw the y coordinate using a temporary variable Y_COR
# that will be incremented while the line is drawn downwards
# from the reference point
#
Y_COR=${Y_REF}
while  true
do
   if [ ${Y_COR} -eq 23 ]
   then
      #
      # reached status line
      #
       break
   fi
   MoveCursor  ${Y_COR}  ${X_REF}
   echo "|"
   Y_COR=`expr ${Y_COR} + 1`
done
#
# save the current location as the
# origin of the graph
#
X_ORIGIN=${X_REF}
Y_ORIGIN=${Y_COR}
#
# draw X coordinate from the origin
#
MoveCursor  ${Y_ORIGIN}   ${X_ORIGIN}
echo "-----5----10---15---20---25---30---35---40---45---50->
${BON}Cycle(s)${BOFF}"
```

```
#
# draw percentage indicators on the y coordinate starting at the
# origin and decrementing Y_COR (currently 23) by two, marking
# the axis with lables showing multiples of 10
#
MoveCursor  ${Y_ORIGIN} ${X_ORIGIN}

Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "10"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "20"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "30"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "40"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "50"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "60"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "70"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "80"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "90"
Y_COR=`expr ${Y_COR} - 2`
MoveCursor  ${Y_COR} ${X_ORIGIN}
echo "100"

}

######################################################################
#
#   Name        : PopulateCpuUsageGraph
#
#   Description : This function populates the cpu usage graph with
#                 cpu usage statistics sampled by the command sar 1 1
#
#   Notes       : 1 The sampling is repeated over the number of
#                   cycles required, which is captured as a command
#                   line argument.
```

```
#
#                  2 CPU usage, the y coordinate, is rounded to the
#                    nearest five percent or multiple of five percent.
#
#                  3 Each point on the x coordinate is one cycle.
#
######################################################################

PopulateCpuUsageGraph ( )

{

trap "HandleInterrupt " $SIGINT  $SIGTERM $SIGHUP

#
# turn reverse video on
#
echo "${RVON}"

NO_OF_REQ_CYCLES=$NO_SAMPLES
NO_CUR_CYCLES=0
#
# move the cursor to origin
#
# MoveCursor ${Y_ORIGIN} ${X_ORIGIN}
#
# re-define the origin to facilitate the drawing the graph
#
X_ORIGIN=`expr ${X_ORIGIN} + 1`
Y_ORIGIN=`expr ${Y_ORIGIN} - 1`

#
# initialize a variable for drawing the x coordinate value
# that is incremented from data from sar sampling
#
X_START=${X_ORIGIN}
#
# sample cpu usage and draw the graph
#
while  true
do
  #
  # re-initialize a variable for drawing the y coordinate
  #
  Y_START=${Y_ORIGIN}
  #
  #  compare cycles
  #
  if [ ${NO_CUR_CYCLES} -ge ${NO_OF_REQ_CYCLES} ]
  then
```

```
          break
     fi
     #
     # run sar the command
     # process the last line, which contains the percent idle figure
     #
     PERCENT_IDLE=`sar 1  1  | tail -1  | awk {'print $5'}`
     PERCENT_USED=`expr 100 - ${PERCENT_IDLE}`
     #
     # work out percent used as a multiple of 5
     #
     YVAL_FOR_PERCENT_USED=`expr ${PERCENT_USED} / 5`
     #
     # adjust it for PERCENT_USED being less than 5
     #
     if [ ${YVAL_FOR_PERCENT_USED} -eq 0 ]
     then
          YVAL_FOR_PERCENT_USED=1
     fi
     #
     #  initialize a counter for drawing cpu usage
     #  on y coordinate
     #
     YVAL_CTR=1
     #
     # start drawing the cpu usage on y coordinate
     #
     while  true
     do
        MoveCursor  ${Y_START} ${X_START}
        echo " "
        #
        # decrement  Y_START
        #
        Y_START=`expr ${Y_START} - 1`
        YVAL_CTR=`expr ${YVAL_CTR} + 1`
        if [ ${YVAL_CTR} -gt ${YVAL_FOR_PERCENT_USED} ]
        then
             break
        fi
     done
     #
     # increment x coordinate value
     #
     X_START=`expr ${X_START} + 1`
     #
     #  increment nunber of cycles
     #
     NO_CUR_CYCLES=`expr ${NO_CUR_CYCLES} + 1`
done
```

```
echo "${RVOFF}"

}

######################################################################
#
# Name           : GetRequiredNumberOfSamples
#
# Description    : The function asks the user for number of cycles
#                  required.
#
# Notes          : 1 The function restricts the number of cycles to
#                    fifty or fewer.
#
######################################################################

GetRequiredNumberOfSamples ( )
{

trap "HandleInterrupt " $SIGINT  $SIGTERM $SIGHUP

while  true
do
  clear
  echo   "Cycle is the number of times the sampling is to repeat"
  echo   "Enter number of cycles (up to fifty) required:\c"
  read  NO_SAMPLES
  case   $NO_SAMPLES  in
   "" )  : ;;
    * ) #
        #  check for digits only
        #
        if [ `expr $NO_SAMPLES + 1 2> /dev/null` ]
        then
              #
              #  check for less or equal to 50
              #
              if [ ${NO_SAMPLES} -le 50  ]
              then
                    break ;
              else
                    DisplayMessage  E "${INVALID_SAMPLE}"
              fi;
        else
                   DisplayMessage  E  "${INVALID_NUMBER}"
        fi ;;
   esac
done

}
```

```
###############################################################################
#
# Name          : main
#
# Description   : The function invokes all other functions
#
###############################################################################

main ()
{

DefineVariables
GetRequiredNumberOfSamples
DrawCpuUsageGraph
PopulateCpuUsageGraph
clear
exit  $SEC
}

#
# execute main
main
```

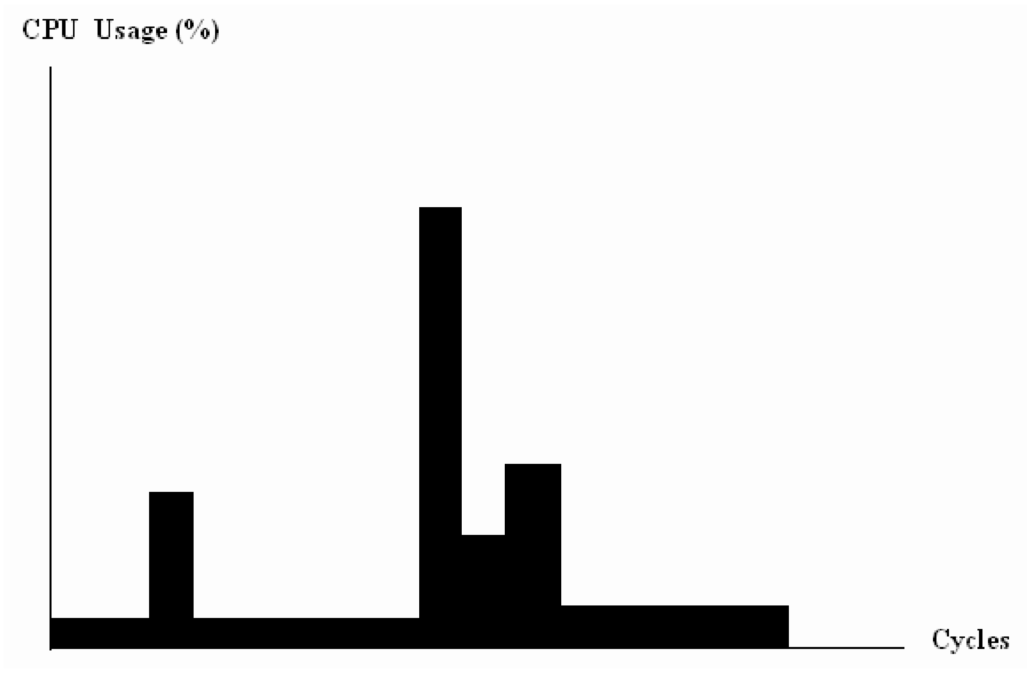

Figure 1: Output chart

SAMPLE OUTPUT

The graph (Figure 1) gives a broad indication of CPU usage and can be used to observe the overall CPU usage of strongly CPU-bound programs. In these circumstances, the graph should depict peaks corresponding to high CPU usage over cycles in which the program runs.

Arif Zaman
Analyst/Programmer
High-Tech Software Ltd (UK)

Running commands and scripts on remote hosts

INTRODUCTION

When a system comprises several machines, it is not unusual for the system administrator to have to run the same system command or script on each machine that makes up the system. For example, to ensure that the system's date is correctly synchronized it's necessary to run the **date** command on every server in the system and check the results.

One quick way of running remote commands is using the **rsh** facility, which allows the execution of commands on remote RS/6000s. For example, to run the **date** command from a local host called *cervino* on a remote host named *lyskamm* use the command **rsh lyskamm date**. This displays the output of **date** run on *lyskamm* on the local system.

While this is a fairly straightforward example, much more useful tasks can be accomplished with this facility. For instance, in an educational environment, where users have login access to multiple servers, **rsh** can be used to synchronize user properties on different machines and to uncover discrepancies in user settings (eg home directory, password expire time, initial shell login, etc). For instance, to control the user *smith* on a remote machine you simply type (as root) **rsh hostname lsuser -f smith**, where **hostname** can be *lyskamm* or any other host in the network.

It's not only root users that benefit from the ability to execute commands remotely – this facility is also useful when looking for users on other nodes, obtaining the load average on different hosts, etc.

BACKGROUND

The mechanism for remotely executing commands is very simple. First of all the user that requires this facility (*smith*) must be defined on every host that allows remote execution – when a request for the execution of a command arrives from the local host (*cervino*) on the remote host (*lyskamm*), the remote host validates the user and checks that he has permission to run commands remotely by checking the configuration file */etc/hosts.equiv* – entries in this file have the following format:

```
# Example of /etc/hosts.equiv file
# To be put on "lyskamm"
# name of the host        user who can execute rsh
  cervino                 +smith.
```

This entry ensures that user *smith* on *cervino* is allowed to perform commands (or perform a remote login) on the local system without supplying a password or other form of validation. User *smith* is therefore considered 'trusted'.

If the entry below is present in the */etc/hosts.equiv* file of host *lyskamm*, it means that every user on *cervino* (if he/she exists) can login to the local system without supplying a password.

```
# Example of /etc/hosts.equiv file
# "name of the host"
cervino
```

Obviously this is a point of weakness for security in your network, especially if users are able to login to a remote system as *root*. For security reason, root authority must never be granted using the */etc/hosts.equiv* file.

Notice that, even if you make sure you've limited access via */etc/hosts.equiv*, there is another mechanism that can be used to logon remotely and even obtain root authority. If a user wants to gain access from *cervino* to *lyskamm* and (vice versa), all he needs to do is create a file named *.rhosts* in his home directory. The format of this file is the same as that of */etc/hosts.equiv*. In this example, user *smith* creates the following file on *lyskamm*:

```
#example of .rhosts file

#to be put under ~smith directory

cervino
```

and an equivalent file on *cervino*:

```
#example of .rhosts file

#to be put under ~smith directory

lyskamm
```

This allows this user to be trusted between *cervino* and *lyskamm*.

This individual method for authenticating remote users is also valid for the root user. Remember that, when the operating system receives a request for a remote command, it first analyses the */etc/hosts.equiv* file, then, if trusting is unsuccessful, it analyses individual *.rhosts* files. The use of *hosts.equiv* is quite involved in itself; if you need to obtain more information about this file, consult the relevant **man** page.

A REAL EXAMPLE

We'll now discuss an example where all the concepts discussed so far are applied.

It is common for AIX systems to work in an environment where hosts are 'clustered' to form larger, more complex domains. When you have lots of users (analysts, programmers, scientific users, etc) that utilize system resources heavily, this allows workload to be divided among different hosts. This difficult set-up can achieved by allowing every

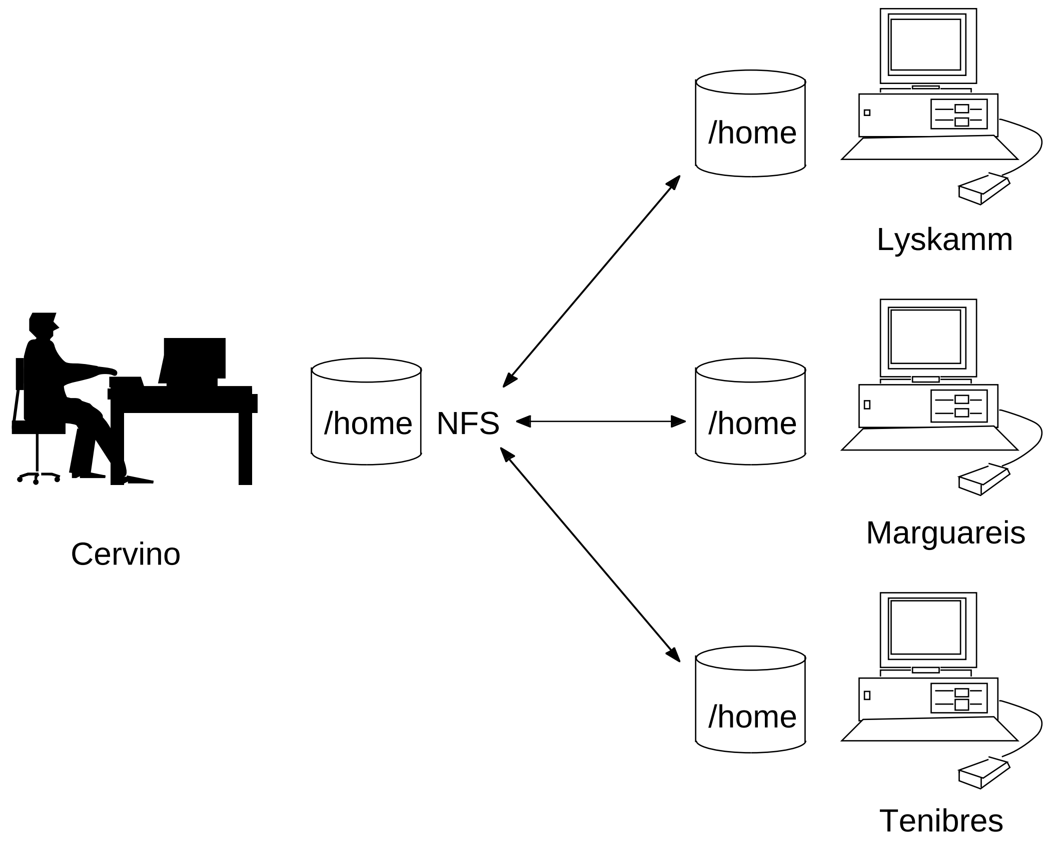

Figure 1: The network set-up

user to login to any host. Obviously this configuration is good as long as users have access to the same environment at every system they use, which means that wherever the user logs in, he or she has access to the same home directory and files.

Figure 1 above illustrates this configuration: the host *cervino* acts as an NFS server, holding users' home directories on one or more filesystems. Other hosts (*lyskamm, tenibres,* and *marguareis*), which collectively comprise the 'client', mount these filesystems remotely, allowing users to work as if files were local. If default NFS settings are used, only the root user of the NFS server has permission to modify and/or deleting files, etc – generic users must be defined both on client and server systems.

Different implementations of this type of system can be far more complex than this, though the concepts remain the same. For instance, it is possible to create groups of servers by area of interest, or to grant access to faster machines only to appropriate users and so on.

Whether the implementation is simple or complex, two main aims have to be achieved:

- Provide users with the same environment on each host
- Allow the system manager to monitor the system efficiently.

To achieve these two results it is necessary to implement the correct trust policy among hosts. To trust generic users it is necessary to edit */etc/hosts.equiv* and add the appropriate entry.

```
# /etc/hosts.equiv file to be put on each machine.

cervino

tenibres

marguareis

lyskamm
```

In this way users can, for instance, execute the command **uptime** on each host, looking for the one with lowest workload. A good way to perform this action is by using the script **rcom** (discussed later).

The trust relationship for the root user is a little bit more complicated: firstly we configure the host *cervino* as the ‘preferential’ machine for system operators. The local root user is then able to execute commands on other hosts. By contrast, the root users of *lyskamm*, *tenibres*, and *marguareis* are not able to perform any functions – either **rlogin** or remote execution – on *cervino*, the NFS server. In this way you can provide root access (with caution of course!) for administrative use on client hosts, and not have to worry about the integrity of users’ data stored in the server.

To complete this task you have to create the following *.rhosts* file in each machine under the root user’s *$HOME* directory:

```
# .rhosts file to be put on each machine under ~root directory

cervino     root
```

That’s all!

MAKING ADMINISTRATORS' WORK EASIER

To enhance the ability to administer such a system we'll use two shell scripts. The first one, **rcom**, allows users to execute **rsh** on a number of hosts without having to repeat the command on each one. For example, issuing the command **rcom df** shows the status of mounted filesystems on each node forming the cluster. To customize the script, change the *HOSTLIST* variable to suit your environment. Typing only **rcom** results in a remote login at every host in the cluster.

If you have to execute complex commands involving pipes or other special characters, remember to enclose your command in double quotation marks ('"'). For example:

```
    rcom "who | grep john"
```

Shows you the host on which user *john* is logged.

As before, you can use this script as a user from any host in the cluster or as root from the NFS server.

RCOM

```
#!/usr/bin/ksh
# To be changed according to your environment
HOSTLIST="cervino lyskamm marguareis tenibres"
tput bold

# this is for setting the "bold" style...
echo "$*"
tput sgr0

# returning in normal mode
echo "will be executed on $HOSTLIST\n"
sleep 3

for HOST in $HOSTLIST
do
echo $HOST
rsh $HOST $*
echo
done
```

Another shell script, named **rscript**, is useful for executing not just one command or a stream of commands on multiple hosts, but a complex, though non-interactive, script or executable file. For example, you could create a script that performs automatic tasks, such as checking filesystem quota, disk usage per user, erasing dummy files, etc, and run it on multiple hosts using this facility (it's my intention to submit scripts along these lines in future). As before, it's convenient to use a single workstation to execute the administrative script on other systems and collect results. To do so, simply type **rscript** ***name_of_script_to_be_executed***. The script distributes the executable file among the hosts contained in *HOSTLIST*, then executes the file on each host, collecting the results and merging them in a single file named */tmp/rscript.out* on local machines. When **rscript** has finished, it also displays results on the screen.

Note that **rscript** creates temporary files in */tmp*, so avoid running more than one instance of this program at the same time and restrict the use of this tool to the root user. In any case, the script uses the **trap** command to avoid leaving temporary files in */tmp* should the script terminate abnormally after receiving an interrupt signal (Ctrl+C).

RSCRIPT

```
#!/bin/ksh
# Modify HOSTLIST according to your environment
HOSTLIST="tenibres cervino lyskamm marguareis"
NAME="$(basename $0)"
PREFIX="/tmp/$NAME"
OUT="$PREFIX.out"

trap i_stop 2

i_stop ()
{
  echo "\nCleaning temporary files ..."
  set -f
  i_do_exec rm -f /tmp/torun*
  set +f
  echo "Done."
  exit
}

i_do_exec ()
{
  for HOST in $HOSTLIST
  do
    echo "$HOST \c"
    rsh $HOST $*
  done
  echo "... Done."
}

i_do_copy_1 ()
{
  echo "Distributing $1 on ..."

  for HOST in $HOSTLIST
  do
    echo "$HOST \c"
    rcp $1 $HOST:/tmp/torun
  done
  echo "... Done."
}

i_do_copy_2 ()
{
  LOCAL=$(hostname)
  echo "\nCopying output from remote hosts:"

  for HOST in $HOSTLIST
  do
    echo "$HOST \c"
    rcp $HOST:/tmp/torun.$HOST $PREFIX.$HOST
  done
  echo "... Done."
}

######################################################################
# Main                                                               #
######################################################################

if ["$1" = "" -o ! -x "$1" -o "$1" = "help" -o "$1" = "-help" -o "$1" =
"-h"]
  then
    echo "Usage:\t$(basename $0) <file> [parameter]\n\twhere the file is
an executable or a script"
  exit
fi

if [ -f $PREFIX* ]
then
  rm -f $PREFIX*
fi

if [ -f /tmp/torun* ]
```

```
  then
    rm /tmp/torun*
fi

i_do_copy_1 $1

echo "\nMaking executables each /tmp/torun"
i_do_exec chmod +x /tmp/torun


echo "\nExecuting command on remote hosts ... (Please Wait)"
i_do_exec "/tmp/torun > /tmp/torun.\$(hostname)"
i_do_copy_2

echo "\nPrinting the list of files created"
find /tmp -name "$NAME.*" -exec ls -l {} \;

echo "\nResults will be stored in $OUT."
sleep 1
> $OUT

for FILE in $(find /tmp -name "$NAME.*" -print)
do
  if [ "$FILE" != "$OUT" ]
  then
    echo "$FILE" >> $OUT
    cat $FILE >> $OUT
    echo "\n---------------------------------------------------------\n"
>> $OUT
  fi
done

ls -la $OUT

echo "Done!"

echo "\nDo you want to see the command's result $OUT? ([Y]/n)"
read ans
if [ "$ans" = "Y" -o "$ans" = "y" -o ! "$ans" ]
then
  more $OUT
fi

i_stop
```

Aiello Maurizio, Cleis Technology (Italy)
Marquez Fabio, Elsag Bailey (Italy)

CC-NUMA

The acronym CC-NUMA has been bandied about in the computer trade press over the past year or so, often with little explanation. The first part of this article describes the CC-NUMA architecture and discusses some of its characteristics, advantages, and disadvantages. The second part then goes on to discuss some of the particular points to consider when evaluating or comparing CC-NUMA systems.

CC-NUMA: THE DEFINITION

CC-NUMA stands for Cache Coherent, Non-Uniform Memory Access. To explain what this means, let's first of all consider a standard Symmetrical Multi Processor (SMP) system (Figure 1).

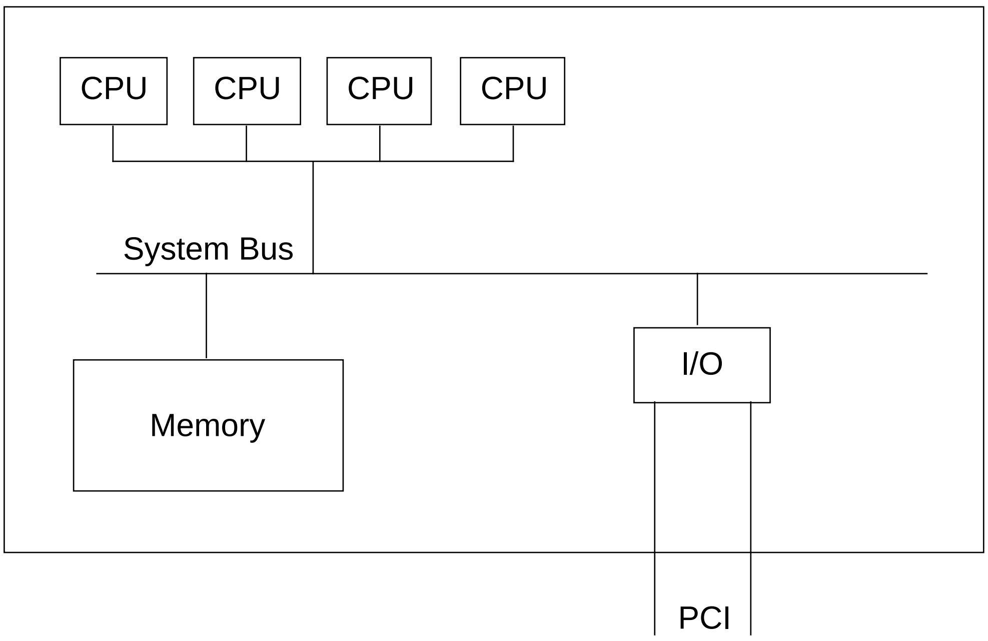

Figure 1: A 'standard' SMP system

In this architecture, all CPUs can access all memory and all I/O. The access time from a CPU to memory and CPU to I/O is independent of CPU. There is a single operating system image which runs on all CPUs.

It is now possible to link several independent SMP systems (I'll refer to such systems as 'modules') via a high-bandwidth, low-latency interconnect, as shown in Figure 2.

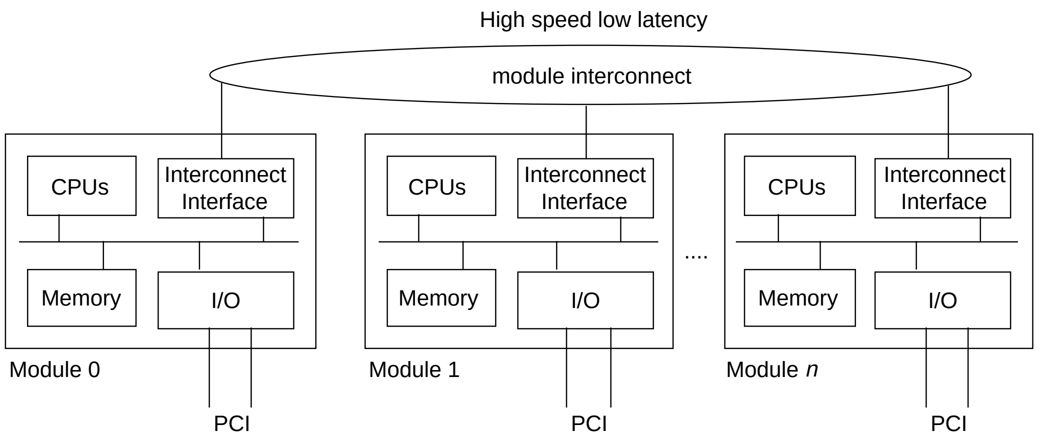

Figure 2: CC-NUMA

As with a 'standard' SMP system, there is only one instance of the operating system, which runs on all CPUs. All system resources (memory and I/O) are visible to all CPUs. There is a global memory map, comprising the memory contained in all modules. For example, the memory address range on module 0 may be from 0-16 GB, and on module 2 from 16 GB to 32 GB, and so on.

However, the access time from a CPU on module 0 to memory on the same module is always going to be less with this architecture than the time to access data on a different module. This is because an access to memory local to the CPU only is via the local system bus, while an access to memory on a remote module has to:

1 Cross the local module's bus to the interconnect interface

2 Cross the interconnect to the remote module's interconnect interface

3 Cross the remote module's bus to memory.

The data must then make the reverse trip to the requesting module.

It is this non-uniformity in memory access time that gives rise to the NUMA part of the CC-NUMA acronym. The ratio below is the 'NUMA factor', which is a measure of the difference in the time taken for local and remote access.

```
remote memory access time
-------------------------
 local memory access time
```

Now that 'NUMA' has been defined, let's look at the 'CC' part of the acronym. CC stands for cache coherent. This means that system hardware is responsible for maintaining data coherency across modules. For example, say CPU 0 on module 0 accesses variable *A*, which is located at a memory address on module 2. CPU 0 now updates this value, for instance by incrementing it by one. If CPU 4 on module 1 now tries to access the same variable, and reads the memory location on module 2, the value stored is invalid or 'stale'. The interconnect hardware detects this situation, intercepts the second memory access, and directs CPU 0 to provide the updated value. This is similar, though not identical, to the technique for maintaining level-1 and level-2 CPU caches coherent on a standard SMP system.

Managing data coherency through hardware has the important characteristic of maintaining the standard SMP programming model. This means that applications written for SMP systems run in an identical manner under CC-NUMA-based systems. In fact, CC-NUMA is just a means of implementing high-order SMP systems. 'Standard' SMP systems are known as UMA systems (for Uniform Memory Access).

It must be pointed out, however, that if data is write-shared by several processes and/or threads, then it's the responsibility of the application to ensure that concurrent accesses to the same data occur in a controlled manner, for example through the use of locks. Cache coherency only ensures that when a CPU reads a memory address it receives the most up-to-date value. If two CPUs modify data and try to write it back to memory without the use of locks or atomic operations, then the probable result is incoherent data.

OPTIMIZATIONS

Now that the basic CC-NUMA architecture has been described, we can now look at a number of optimizations that are possible. All of these optimizations try to place data close to the CPUs that access it (that is, on the same module). There are two complementary approaches:

1 Place or move the data close to the CPU

2 Schedule the executable thread on a CPU close to the data that it will access.

For the following discussion I will use the following terms:

- *Local memory* is memory that is on the same module as the CPU that accesses it.
- *Remote memory* is memory that is on a different module from the CPU that accesses it.
- The *home module* is the module where a given memory address is located.
- The *owner module* is the module that has an updated value of a given memory address.

The last two items deserve some clarification. Clearly a given address in real memory must reside on a given module. For example address *A* may be on module *x*. In this case *x* is said to be the home module. When a CPU on a module, say module *y*, requests the data value held at address *A*, then module *y* becomes the owner module. Thus the home module and owner module may be one and the same, but are not necessarily so.

Optimizations are important for CC-NUMA and can have a major impact on system performance. The NUMA factor is typically in the range of three to twenty, depending on the nature of the interconnect (hardware, communications, and coherency protocol). This means that accessing remote memory can take between three and twenty times longer than accessing local memory. Additionally, without optimization, a two-module system on average accesses remote memory 50% of the time, while an eight-module system accesses remote memory 87.5% (7/8) of the time. The need for (and benefits of) optimization increase with NUMA factor and number of modules.

As an example of the complexity of the problem, consider the following I/O scenario. As previously stated, all system resources are visible to all processors. With a three-module system, as shown in Figure 3, it is possible for a CPU on one module to initiate an I/O operation (for example a disk read) on a device on a different module. During such an operation, the disk controller performs a DMA (Direct Memory Access) operation to a buffer somewhere in memory.

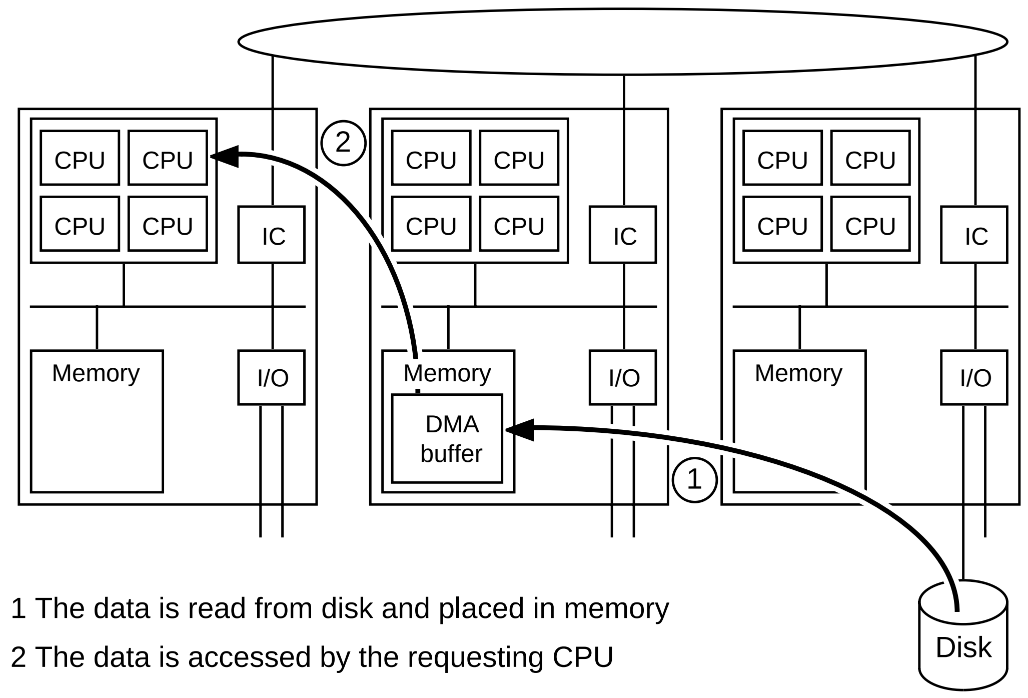
CPU
CPU
CPU
CPU
IC
Memory
I/O
2
DMA buffer
1
Disk
1 The data is read from disk and placed in memory
2 The data is accessed by the requesting CPU

Figure 3: Three module I/O scenario

Suppose the buffer is physically located on a module other than the one in which the CPU or the disk is located (the buffer could also be split across two or more modules). Now the DMA from disk to memory takes place across the interconnect to the module containing the buffer. When the I/O operation is complete, the disk controller signals the requesting CPU with an interrupt, which causes it to read the buffer, requiring a second transmission of data across the interconnect from the module containing the buffer to the one containing the CPU. Optimizations can be made both at the system hardware level and at the operating system level. These are discussed below.

HARDWARE OPTIMIZATIONS

L3 cache

In the same way that individual CPUs on an SMP system have their own level-1 and level-2 cache subsystems, it is possible to implement a level-3 cache for each module. This is shown schematically in Figure 4. In this case data is accessed by all processors on the module. The cache holds only remote memory data. There are no local memory addresses present in the L3 cache.

The use of L3 caching means that, while the initial remote memory access incurs the full 'NUMA Factor penalty', subsequent access to the same data by any processor on the module has the same latency as local access, provided the data is still resident in cache.

All caches use a hashing algorithm to map physical memory addresses

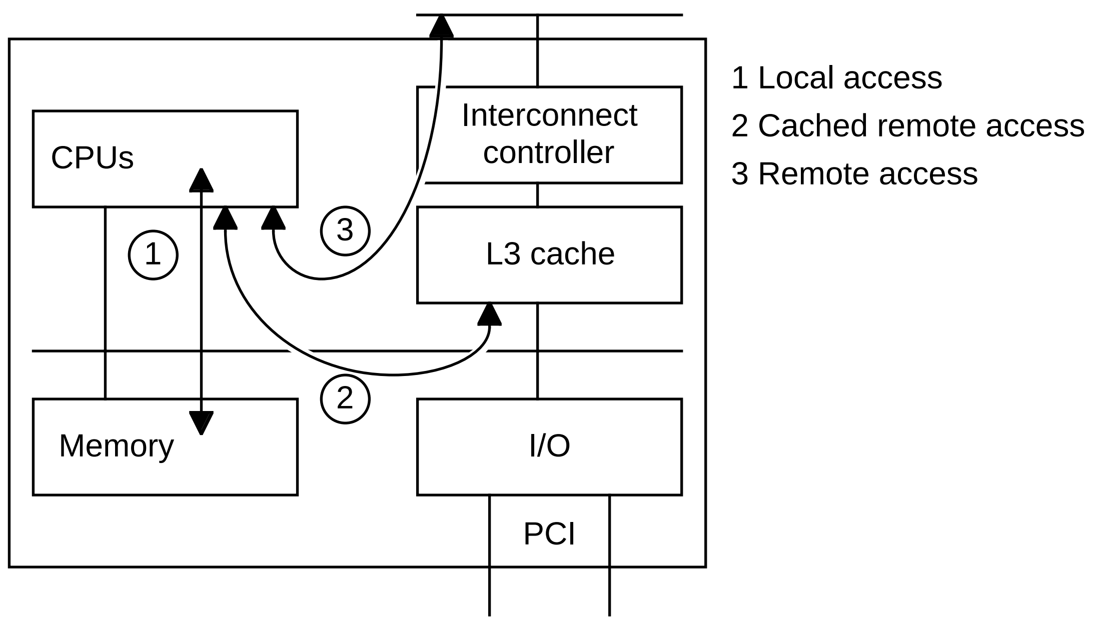

Figure 4: CC-NUMA with L3 cache

to cache lines. As cache is always smaller than the area of memory it caches, several memory addresses map to the same line. A collision occurs when cached data that's in use by one CPU is evicted by another one requesting another memory location that maps to the same cache line. To be efficient, L3 cache should be large and (preferably) use an associative organization to reduce the number of collisions on individual cache lines.

For example, it's not unusual for individual modules to support at least 4 GB of physical memory. In an eight-module system each module has 28 GB of remote memory. With a 32 MB non-associative L3 cache, a cache line collision occurs about once every 900 accesses, on average. In addition, the working set (that is, the set of regularly accessed memory addresses) of today's operating systems is probably in the order of 32 MB. With a 32 MB L3 cache, caching application code and data (such as a database) is done at the expense of evicting operating system code from the cache, thus reducing system performance. For such a configuration running enterprise type applications a cache size of at least 256 MB is required.

Cache coherence protocol

The efficiency of the cache coherence protocol has a large impact on the NUMA Factor. On a traditional SMP system, cache coherency is generally maintained by a 'snoopy' protocol, in which each CPU examines addresses being transmitted on the address bus. If a CPU detects a request for a memory location for which it has an updated value, it provides the value to the requesting processor.

This technique works when CPUs are located physically close to each other and when the number of CPUs is limited. To adopt the same technique for intermodule cache coherence would be prohibitively expensive in terms of bandwidth and latency. Because of this, most CC-NUMA implementations use a directory-based cache coherence protocol, whereby only modules that are affected by the coherence operation take part in the transaction.

Intervention

When a module (or, more accurately, a CPU on a module) tries to read remote data and the home module is not the owner module, what normally happens in a directory-based cache coherence protocol is that the home module sends a request to the owner module asking for an updated version of the data. Once this is received, the home module replies to the original requesting module. Using a technique known as ‘intervention’ it’s possible to eliminate one of these data transfers. Using intervention, when the home module receives a request for data, instead of asking the owner to send it the updated value, it asks the owner to send the updated value directly to the requester. This technique can significantly reduce latency and the NUMA factor.

SOFTWARE OPTIMIZATIONS

The impact of the NUMA Factor

As discussed above, the NUMA Factor typically varies between three and twenty. With a NUMA Factor of five or less, reasonable scalability can be achieved without compromising performance or requiring complex operating system modifications. This means that the system performance increases fairly linearly with a gradient of about one with each additional module (up to a limit). However, with a NUMA Factor of five or more, it’s necessary to modify the operating system so that data and executable threads that operate on them are close together. Two operating system components have a role to play in such an arrangement – the Virtual Memory Manager (VMM) and scheduler.

VMM allocation strategies

When an application makes a request for memory, for example via the *malloc()* system call, the VMM reserves space to satisfy this demand in physical memory. In non-NUMA system, the VMM usually uses a least-recently-used (LRU) algorithm to decide where to place the allocated memory. However, in NUMA systems, the objective is to keep data close the CPU that is going to use it, so one of the first modifications to make is to ensure that, when an application requests virtual memory, the VMM tries to allocate physical memory on the same module as the CPU on which the application runs.

This simple modification can significantly reduce the number of remote data accesses and increase performance accordingly. There are occasions, however, when it fails. For example, most database implementations have an initialization phase during which the various database buffers are allocated. At the end of this phase, a number of ‘worker threads’ and/or processes are created to implement the database function. These worker processes, which are independent of the initialization process, are likely to be scheduled on any CPU on any module, and consequently the database buffers they use may well be on a different module to threads.

This leads us to another optimization technique: page migration.

Page migration

As described above, it’s possible that the VMM’s allocation strategy will not always be optimal for NUMA. In addition, threads and processes may migrate from one module to another in order to optimize CPU usage. Under these circumstances the number of remote accesses may be very high. It’s possible for the operating system to detect that one or more CPUs of a given module are systematically accessing remote memory and to request that the VMM physically moves the memory pages concerned to the module making the accesses. However, care must be taken to avoid ‘ping-ponging’ pages between two or more modules.

Page replication

It is possible for several modules to access the same set of memory addresses. In this case page migration doesn’t work. One solution to this problem is for each module to make a copy of the data and keep it locally. This technique is very effective for read-only data, such as kernel or application executables, but generates a large amount of coherence traffic in the event of write operations.

Scheduler affinity

In today’s modern operating systems, the schedulable entity is usually an executable thread. Each thread is given a time slice of CPU; once its time is up, it’s evicted and another thread is scheduled on the processor. In a NUMA system, executing threads bring data with them into the L3 cache. It makes sense, therefore, to try to make the most of this data, so the next time that the thread is eligible for execution, the scheduler should try to schedule it on a CPU on the same module. This is known as 'soft affinity'.

A variation on this theme is 'hard affinity', whereby the application or system administrator binds a thread or process either to an individual CPU or a module. By binding a thread, the thread can run on only that module or CPU, even if there are no available CPUs on that module and idle ones on other modules.

This article concludes in next month's issue of AIX Update.

Jean-Paul Weber (France)

Using signals to kill a process

In Unix, a signal is sent to a process when an event exterior to the process occurs to which the process must respond. The simplest example of a signal is 'hang up' (*SIGHUP*). When a user is logged on at a remote terminal, the data line can hang up for a number of reasons. Line problems, modem problems, power loss at the remote terminal, or deliberately or accidentally turning off the remote terminal all result in a hang up signal. Unix keeps track of which processes are being run by which terminal, and when a terminal hangs (drops its connection) the operating system sends a *SIGHUP* to all the processes that were launched from that terminal.

A process has three options when it receives a *SIGHUP* signal:

- The process can stop executing immediately, which is the default action.
- The process can catch (trap) the signal, ignore it, and continue executing.
- The process can catch the signal and carry out some other programmed reaction. For example, it could close all open files, display a warning message, and then exit.

THE KILL COMMAND

The kill command can be used to send signals to a running process. Its syntax is:

```
    kill -<signalNumber> <jobNumber>
```

The hang up signal has *signalNumber* 1, so the command **kill -1 1234** sends the hang up signal to job *1234*, in the same way as if the user had turned off the terminal while logged on.

But before you begin killing processes and possibly causing havoc, it's necessary to understand what signals do and how programs handle them – there are clean and less than clean methods of killing a process!

TRAPPING SIGNALS IN SHELL SCRIPTS

Enter the following shell script and call it **alive.sh**.

ALIVE.SH

```
# alive.sh

while true
do
    echo "I'm alive!"
    sleep 5
done
```

Make it executable with:

```
    $ chmod +x alive.sh
```

Run it in the background by adding an ampersand (&) after its name:

```
    $ alive.sh & [1] 5678 $
```

5678 is the job number, or process ID, of the **alive.sh** command now running in the background. Despite the fact that the command runs in the background, the message '*I'm alive!'* still appears on your terminal every five seconds. Kill it by sending it the *SIGHUP* signal.

```
    $ kill -1 5678
```

In the following listing, our **alive.sh** command has been modified to trap the *SIGHUP* signal. The trap's syntax is:

```
    trap functionName signalNumber
```

In our example the function *signal01* is called when our shell script receives the *SIGHUP* signal.

MODIFIED SCRIPT

```
# alive.sh

function signal01 {
    echo "Received signal 1 (SIGHUP)"
}

trap signal01 1

while true
do
    echo "I'm alive!"
    sleep 5
done
```

If we now execute the **alive.sh** shell script, appending an ampersand to the command, and attempt to kill it with **-1**, we see the following output at our terminal:

```
$ alive.sh &
[1] 7788
$ I'm alive!
I'm alive!
I'm alive!
kill -1 7788
$ Received signal 1 (SIGHUP)
I'm alive!
I'm alive!
```

The trap catches *signal 1* and simply displays a message and continues. You can stop the program by sending a different signal such as *signal 2* (*SIGINT*) with the command **kill -2 7788**.

This technique is used in more complex shell scripts. If the script is in the middle of an important or complex calculation or action, rather than just ‘dropping dead’, the script finishes its current action and carries out other housekeeping, such as ensuring that open files are closed, before terminating.

SIGKILL: A SIGNAL THAT CANNOT BE IGNORED

Signal 9 (*SIGKILL*) is unlike other signals in that it cannot be trapped.

Sending *signal 9* to a process means that the operating system must immediately kill the process. The advantage of *signal 9* is that the program cannot trap it and ignore it; the disadvantage is that the program cannot intercept it and perform an orderly shut down.

Using the **kill -9** on a database or similar process can be disastrous. It's important first to attempt to kill such a process with *SIGHUP* or *SIGINT* before resorting to the deadly *SIGKILL*. Note that there are instances when even **kill -9** won't kill a process. An example of this is when external devices, such as tape drives, are involved.

TRAPPING SIGNALS IN A C PROGRAM

The following program (**alive.c**) performs a similar function to our shell script **alive.sh**. Again the *SIGHUP* signal is trapped to display a message.

SAMPLE C PROGRAM

```
#include    <stdio.h>
#include    <signal.h>

void
signal01() {
    (void) printf("Received signal 1 (SIGHUP)\n");
}

main()
{
    (void) signal(SIGHUP, signal01);

    while ( 1 ) {
        (void) printf("I'm alive\n");
        (void) sleep(5);
    }
}
```

SOME COMMON SIGNALS

- *1* *SIGHUP* ('hang up') is the result of a phone line or terminal connection being dropped.
- *2* *SIGINT* ('interrupt') is generated from the keyboard, for instance by pressing Ctrl-C.

- *3* *SIGQUIT* (‘quit’) is generated from the keyboard, usually by a Ctrl-\ or Ctrl-Y. To find out which, type **stty -a** and press enter. In the listing you will find ‘quit=^\’, or ‘quit=^Y’, or something similar. A *SIGQUIT* often causes a core file to be created, containing a copy of your working memory.
- *15* *SIGTERM* (‘software terminate’) is often used to terminate a program. Using the **kill** command without a signal number causes it to send its default, *signal 15*, to the job. This is a good first step when trying to kill a process.

A CLEAN KILL

Time and effort is necessary to code a trap for a signal into a program. This means that, when a trap has been coded in a program, it’s been done for a good reason. If the program can simply die without performing any cleanup, then why go to the trouble of including a trap? That is why it’s a good idea to try *signal 15*, *signal 1*, and *signal 2* before resorting to *signal 9*.

I use the following shell script to kill processes cleanly.

SCRIPT TO KILL PROCESSES

```
#!/bin/sh
#
# kill.sh  --  kill "cleanly"

for pid in $*
do
    kill    $pid
    kill -1 $pid
    kill -2 $pid
    kill -9 $pid
done
```

Note that killing a process that has already been killed results in a harmless error message. Another good idea is to try to kill processes in an orderly manner using ‘weaker’ signals before trying to massacre them using *SIGKILL*.

AIX Specialist (Switzerland) © Xephon 1998

AIX news

Sybase has announced Replication Server version 11.5, which features better management facilities and support for more than 25 different types of data source, including application packages. A new replication management framework simplifies set up and synchronization of data between systems, including a new graphical replication manager tool.

Replication Server can now be managed by systems management tools from Tivoli, BMC and Compuware, with improved warm-standby features.

Out now for AIX, prices start at US$2,695 for two to eight concurrent users.

For further information contact:
Sybase Inc, 6475 Christie Avenue, Emeryville, CA 94608, USA
Tel: +1 510 922 3500
Fax: +1 510 658 9441
Web: http://www.sybase.com

Sybase UK Ltd, Sybase Court, Crown Lane, Maidenhead, Berkshire Sl6 8QZ, UK
Tel: +44 1628 597100
Fax: +44 1628 597000

* * *

Rational Software has announced new releases of its Unix development tools, including Rational Rose 98 for visual modelling. The package provides language-independent enterprise development capabilities, and also integrates with ClearCase, Rational's software configuration management package. It also boasts additional support for Unified Modelling Language v1.1. Rational Rose is available for AIX at USD6,000.

For further information contact:
Rational Software, 18880 Homestead Road, Cupertino, CA 95014, USA
Tel: +1 408 862 9900
Web: http://www.rational.com

Rational Software, Olivier House, 18 Marine Parade, Brighton BN2 1TL, UK
Tel: +44 1273 624814
Fax: +44 1273 624364

* * *

IBM has announced a high-performance compiler for Java running on AIX (and also OS/2, Windows 95, and Windows NT systems), which compiles Java bytecode into optimized platform-specific native code. This is significantly faster than bytecode executed in a JVM/JIT environment. The degree of performance improvement depends upon the application. The current beta release supports a subset of the JDK1.1.1 APIs.

IBM has also let it slip that its RS/6000 server line is to become the first operating system to receive Virtual Private Network certification by the International Security Association, the ICSA. The certification is for AIX 4.3.1, which is already the holder of Germany's E3/F-C2 certification (though not the DoD's C2 certification).

For further details contact your local IBM representative.