56 AIX update

June 2000

In this issue

3 Version control for files on AIX servers

9 Check mail utility

28 RAID and AIX

38 ZeroFault – a memory debugging tool for AIX

52 AIX news

Published by

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

North American office

Xephon/QNA
Post Office Box 350100, Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290

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

Disclaimer

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

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1995 issue, are available separately to subscribers for £16.00 ($23.00) each including postage.

AIX Update on-line

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

Printed in England.

Version control for files on AIX servers

The company I work for has about 200 remote AIX servers, and we are responsible for carrying out software distribution to those servers. One of the most important aspects of this task is to keep all servers at the same version level. For this purpose, I have written the script **compare.sh** (see the listing below). For more than two years this script has successfully carried out version control on our remote servers.

compare.sh needs three pieces of information, which are all entered interactively once the script is running. These are the reference server's name, the 'plan' name (the plan is a file that includes the names of all servers to be compared with the reference server), and the parallel compare (**rexec**) limit number. Basically, we need a few fixed directories to be compared and the differences to be written to a file (*server_name.out*) for each server (note that, as the directory names are hard-coded, you'll need to change them to suit your installation's own requirements). The name, date, and size of applications are the criteria by which we determine whether an application is different from its equivalent one on the reference server. For example, let's check the contents of directory *101.out* on server *101* with reference server *918*. The output of the script is as follows:

```
101 77792 21 Mar 17:17 /u/winappsnt/gturig00.app
101 130718 28 Feb 17:15 /u/winappsnt/gturkg00.app
101 70638 18 Jan 17:02 /u/winappsnt/gtursg00.app
101 78382 14 Jan 15:58 /u/winappsnt/gturpg00.app
101 44168 14 Jan 15:56 /u/winappsnt/gturag00.app
101 0 12 Oct 14:46 /u/winappsnt/futbank.txt
101 3000 26 Jan 13:25 /u/vggen/load/gtcell.tab
101 190399 21 Mar 17:11 /u/vggen/load/turici0.ibmcpp
101 4418 21 Mar 17:11 /u/vggen/load/turici0.bnd
101 188717 26 Jan 13:26 /u/vggen/load/turmcu0.ibmcpp
101 4817 26 Jan 13:26 /u/vggen/load/turmcu0.bnd
101 184222 18 Jan 16:55 /u/vggen/load/turscu0.ibmcpp
101 4817 18 Jan 16:55 /u/vggen/load/turscu0.bnd
101 184502 18 Jan 16:55 /u/vggen/load/turkcu0.ibmcpp
101 4817 18 Jan 16:55 /u/vggen/load/turkcu0.bnd
101 194021 14 Jan 15:58 /u/vggen/load/turpci0.ibmcpp
101 8819 14 Jan 15:58 /u/vggen/load/turpci0.bnd
101 68002 14 Jan 15:57 /u/vggen/load/turpcd0.ibmcpp
101 175449 14 Jan 15:55 /u/vggen/load/turkci0.ibmcpp
101 3633 14 Jan 15:55 /u/vggen/load/turkci0.bnd
```

As can be seen above, applications on server *101* whose version differs from that on the remote server are listed. The listing comprises the server name, the size of the mismatched application, its date, and its path. Note that all of the applications above either do not exist on our reference server, or have a different size or date from ones on the reference server. If a **rexec** on a remote server fails, the names of servers that are not available are listed in a file called *closed.out*. There are several ways of displaying these output files. I programmed a user-friendly tool in Delphi to see the output in an NT environment using NFS.

Note the use of the continuation character, '➤', in the code below to indicate a formatting line break that's not present in the original code.

COMPARE.SH

```
#!/bin/ksh
# By Adnan Akbas Jan'98
# A version control script that compares files in specific directories
# in AIX servers with ones on a reference server.

cnt=0
plan=plan_all
max_limit=15

#####################################
# main program                      #
#####################################

#############################################################################
# checking if there is a compare running now...                             #
#############################################################################

ls -l /u/ftpuser/compare/aixcomp???.msgl 2>/dev/null | wc -l | read
➤  msgl_num
if [[ $msgl_num -gt 0 ]] ; then
   clear
   ls /u/ftpuser/compare/aixcomp???.msgl
   print " "
   print "AIX compare script currently running..."
   print " "
   print "Do you want to start a new compare?"
   read key?" y / n "
   if [[ $key = y* || $key = Y* ]] ; then
       rm -f /u/ftpuser/compare/aixcomp???.msgl
   else
```

```
      print "Wait till script terminates and try again..."
      exit 3
  fi
fi

#############################################################################
# asking the reference server ....                                          #
#############################################################################

clear
print -n "Enter your reference server..."
read sube1 junk

#############################################################################
# asking the plan...                                                        #
#############################################################################

# The 'plan' is a file with a list of the servers to be compared
clear
print \\nDefault plan = $plan
read chk_plan?"To change press 1; to accept press ENTER."
if [[ $chk_plan -eq 1 ]] ; then
   print \\nPlans to use :\\n
   ls plan*
   print \\n
   read plan?"Enter the plan name :"
   if [[ ! -s $plan ]] ; then
      print \\nThis plan does not exist or is empty. Please check.
      print \\nAfter checking the plan, re-start the script.\\n
      exit 1
   fi # check that plan exist and is not empty.
fi # check whether plan has changed.

cat $plan | wc -l | read subelist_max

#############################################################################
# asking parallel rexec limit...                                            #
#############################################################################

clear
print \\nDefault maximum parallel compare limit is $max_limit
read chk_com?"To change press 1; to accept press ENTER."
if [[ $chk_com -eq 1 ]] ; then
   read max_limit?"Enter parallel compare limit: "
   if [[ $max_limit -lt 1 || $max_limit -gt $subelist_max ]] ; then
      print \\nParallel compare limit must be between 1-$subelist_max.
      print \\nAfter checking the limit, re-start the script.\\n
      exit
   fi # check whether rcp is meaningless.
fi # check whether rexec has changed.
```

```
################################################################################
# clear logs?                                                                  #
################################################################################

print ; print
read yanit?"Do you want to delete compare (*.out) logs? (y/n) "
if [[ $yanit = y* || $yanit = Y* ]] ; then
    print "Deleting compare (*.out) logs..."
    rm -f /u/winapps/dsyu/compare/*.out
    rm -f /u/ftpuser/compare/*.out

print "Compare logs deleted."
    read anykey?"To continue, press ENTER..."
else
    print "Compare logs are not deleted."
    read anykey?"To continue, press ENTER..."
fi

clear
print
print "Please wait..."
print

################################################################################
# taking data from the reference server...                                     #
################################################################################

# Alphabetical grouping needed for "ls -lt" not to return a
# "The parameter list is too long" error.

rexec an${sube1} 'ls -lt /u/winappsnt/[a-e,A-E]*'
➤  >/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/winappsnt/[f-m,F-M]*'
➤  >>/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/winappsnt/[n-z,N-Z]*'
➤  >>/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/vggen/load/[a-e,A-E]*'
➤  >>/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/vggen/load/[f-m,F-M]*'
➤  >>/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/vggen/load/[n-z,N-Z]*'
➤  >>/u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/vggen/forms/[a-m,A-M]*'
➤  >> /u/ftpuser/compare/${sube1}1.out
rexec an${sube1} 'ls -lt /u/vggen/forms/[n-z,N-Z]*'
➤  >> /u/ftpuser/compare/${sube1}1.out

# take the required fields from the output (size date name)

awk '{print $5, $6, $7, $8, $9}'
```

```
➤  /u/ftpuser/compare/${sube1}1.out>/u/ftpuser/compare/${sube1}.out

################################################################################
# function doit                                                                #
################################################################################

function doit {

   msgl_file=/u/ftpuser/compare/aixcomp${sube2}.msgl
   touch $msgl_file
   print "$cnt / $subelist_max - comparing on $sube2..."

# First rexec returns a code that determines whether the server
# is available. If the server is off-line, then the server name
# is written to the file 'closed.out'.

# Alphabetical grouping is needed for 'ls -lt', otherwise it returns a
# "The parameter list is too long" error.

rexec an${sube2} 'ls -lt /u/winappsnt/[a-e,A-E]*'
➤  >/u/ftpuser/compare/${sube2}2.out 2> /dev/null

   if [[ $? = 0 ]] ; then
      rexec an${sube2} 'ls -lt /u/winappsnt/[f-m,F-M]*'
➤  >>/u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/winappsnt/[n-z,N-Z]*'
➤  >>/u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/vggen/load/[a-e,A-E]*'
➤  >>/u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/vggen/load/[f-m,F-M]*'
➤  >>/u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/vggen/load/[n-z,N-Z]*'
➤  >>/u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/vggen/forms/[a-m,A-M]*'
➤  >> /u/ftpuser/compare/${sube2}2.out 2> /dev/null
      rexec an${sube2} 'ls -lt /u/vggen/forms/[n-z,N-Z]*'
➤  >> /u/ftpuser/compare/
      ${sube2}2.out 2> /dev/null

# taking the required fields of the output (size date name)

    awk '{print $5, $6, $7, $8, $9}'
    ➤  /u/ftpuser/compare/${sube2}2.out
    ➤  >/u/ftpuser/compare/${sube2}.out

################################################################################
# compare                                                                      #
################################################################################

      diff -w /u/ftpuser/compare/${sube1}.out
```

```
      ➤  /u/ftpuser/compare/${sube2}.out |
      grep -E "^>|^<" | grep -vE "^> total|^< total" | awk
      ➤  'gsub("<",'$sube1') gsub(">",'$sube2')' >
      ➤  /u/winapps/dsyu/compare/${sube2}.out

      rm -f /u/ftpuser/compare/${sube2}*.out
   else
      print "$sube2" >> /u/winapps/dsyu/compare/closed.out
   fi
   rm -f $msgl_file
}

###############################################################################
# function Wait                                                               #
###############################################################################

function Wait {
ls -l /u/ftpuser/compare/aixcomp???.msgl 2>/dev/null | wc -l |
➤  read comp
while (( $comp > $max_limit )) ; do
  print "ZZZzzz parallel compare is $comp"
  sleep 20
  ls -l /u/ftpuser/compare/aixcomp???.msgl | wc -l | read comp
done
}

for sube2 in `cat /u/ftpuser/${plan}` ; do
   Wait
   let cnt+=1
   doit &
done

###############################################################################
# Writing the parallel compare number on the screen ...                       #
###############################################################################

ls -l /u/ftpuser/compare/aixcomp*.msgl 2>/dev/null | wc -l |
➤  read active_comp
print " "
while [[ $active_comp -gt 0 ]] ; do
   print "$(tput cuu1) $active_comp compare is working now."
   sleep 2
   ls -l /u/ftpuser/compare/aixcomp*.msgl 2>/dev/null | wc -l |
   ➤  read active_comp
done
```

Adnan Akbas
System Programmer
Pamukbank (Turkey)

Check mail utility

INTRODUCTION

This utility checks for new mail at a given interval and notifies the recipient of any that's arrived. Notification is via a reverse video window that appears in the middle of the screen at all terminals to which the recipient is connected. While the service is also available when the user is in a shell, the notification is then only a string that appears on the command line. This utility was developed to replace the existing mail notification mechanism.

UTILITY COMPONENTS

1 **chkmd.sh** – check mail daemon script

2 **schkmd.sh** – start check mail daemon script

3 **kchkmd.sh** – kill check mail daemon script

4 **newmail.c** – a C program that displays a reverse video window on the screen that notifies the user of new mail.

Note the use of the continuation character ('➤') in the code below to indicate a formatting line break that's not present in the origianal code.

LISTING OF CHKMD.SH

```
#############################################################################
# Name        : chkmd.sh (check mail daemon)
#
# Description: The script checks for new mail sent to a user,
#              checking at regular intervals, and notifies the
#              user of new mail using a reverse-video window.
#
# Notes       : 1. The script must be started by schkmd.sh
#               2. The script contains the following functions:
#                  o main
#                  o ProcessMail
#                  o InitializeVariables
#                  o InitializeMailList
#                  o CheckMail
#                  o HandleInterrupt
```

```
#                 o NotifyuserOfNewMail
#                 o ProcessMail
#                 o ProcessExit.
################################################################################
# Name       : InitializeVariables
#
# Description: The function initializes all variables.
################################################################################
InitializeVariables ()
{
INDEX=1                     # working index
MAIL_DIR="/var/spool/mail"
MLI=1                       # index to mail list array
MAIL_LIST_ARRAY[$MLI]=""    # mail list array holding recipient's name
                            # and latest message id
TEMP_FILE="/tmp/chkmd_$$.tmp"
# return codes
TRUE=0
FALSE=1
# define signal
SIGTERM=15; export SIGTERM
SIGINT=2; export SIGINT
}
################################################################################
# Name       : HandleInterrupt
#
# Description: The function calls ProcessExit.
################################################################################
HandleInterrupt ()
{
DATETIME=`date "+%d/%m/%Y %H:%M:%S"`
echo "chkmd.sh:ERROR:Program terminated on ${DATETIME}" >> ${LOG_FILE}
ProcessExit $FEC
}
################################################################################
# Name       : ProcessExit
#
# Description: The function implements a graceful exit.
################################################################################
ProcessExit ( )
{
EXIT_CODE="$1"
rm -f ${TEMP_FILE}
#
# write log message
#
echo "chkmd.sh:INFO:Exiting the daemon" >> ${LOG_FILE}
exit $EXIT_CODE
}
################################################################################
# Name       : Processkey
```



```
#
# Description: The function checks that the daemon was started
#              by the program schkmd.sh.
#
# Returns    : $TRUE or  $FALSE
#
# Notes      : 1. The variable $KEY, which contains the parent PID,
#                 is exported by schkmd.sh. If the daemon chkmd.sh
#                 is not started by schkmd.sh, the $KEY and $PPID
#                 don't match.
################################################################################
ProcessKey ( )
{
trap "HandleInterrupt " $SIGTERM  # process this signal
#
# exported variable $KEY and the parent process id of this process
# must match
#
if [ "${KEY}" != $PPID ]
then
    return $FALSE
else
    return $TRUE
fi
}
################################################################################
# Name       : NotifyUserOfNewMail
#
# Description: The function notifies a user of new mail.
#
# Input      : 1. User Id
#
# Returns    : $TRUE or
#              $FALSE
#
# Notes      : 1. The function uses the following command to notify
#                 the user:
#
#                     newmail > ${TERMINAL_ID}
#
#                 where 'newmail' is an executable of a c program in
#                 /usr/bin written using the curses library.
################################################################################
NotifyUserOfNewMail ( )
{
trap "HandleInterrupt " $SIGTERM  # process this signal
#
# assign parameter
#
P_USER="$1"
#
```

```
# process users who are currently connected to the system
#
w -h | awk {'print $1 "  " $2'} | while read USER TERMINAL_ID
do
    if [ "${USER}" = "${P_USER}" ]
    then
        /usr/bin/newmail > /dev/${TERMINAL_ID}
    fi
done
return $TRUE
}
###############################################################################
# Name       : InitializeMailList
#
# Description: Initializes the array $MAIL_LIST_ARRAY[] with user-id,
#              latest message-id, and user processed flag with a
#              value of N.
#
# Returns    : $TRUE or $FALSE.
###############################################################################
InitializeMailList ( )
{
trap "HandleInterrupt " $SIGTERM        # process this signal
#
# switch to main mail directory
#
cd ${MAIL_DIR}
#
# get all the file names
#
ls -1 > ${TEMP_FILE}
#
# retrieve latest mail ids and store with corresponding mail users
#
MLI=1
cat $TEMP_FILE | while read MAIL_FILE_NAME
do
    MSG_ID=`tail ${MAIL_FILE_NAME} | grep "Message-Id" |
    ➤  awk {'print $2'}`
    #
    # user id:message id:remove flag(Y or N)
    #
    MAIL_LIST_ARRAY[$MLI]="${MAIL_FILE_NAME}:${MSG_ID}:N"
    MLI=`expr $MLI + 1`
done
return $TRUE
}
###############################################################################
# Name       : ProcessMail
#
# Description: Processes mail to establish whether is is new. If yes,
```

```
#               it invokes NotifyuserOfNewMail().
#
# Input       : 1. User Id
#               2. Mail Id
#
# Returns     : $TRUE or $FALSE
#
# Notes       : 1.The function calls the following functions:
#                 o NotifyUserOfNewMail.
##############################################################################
ProcessMail ( )
{
trap "HandleInterrupt " $SIGTERM            # process this signal
#
# Assign parameters
#
P_USERID="$1"
P_NEW_MSG_ID="$2"
#
# Process the input details againt $MAIL_LIST_ARRAY[] array
#
INDEX=1
while [ "${MAIL_LIST_ARRAY[$INDEX]}" != "" ]
do
    MAIL_USER_ID=`echo ${MAIL_LIST_ARRAY[$INDEX]} | cut -d':' -f1`
    LAST_MSG_ID=`echo ${MAIL_LIST_ARRAY[$INDEX]} | cut -d':' -f2`
    if [ "${MAIL_USER_ID}" = "${P_USERID}"  ]
    then
        #
        # Update the mail list
        #
        MAIL_LIST_ARRAY[$INDEX]="${P_USERID}:${P_NEW_MSG_ID}:Y"
        #
        # Found the recipient; now compare the message id
        #
        if [ "${LAST_MSG_ID}" = "${P_NEW_MSG_ID}" ]
        then
            #
            # No mew mail received
            #
            return $TRUE
        else
            #
            # New mail received
            #
            NotifyUserOfNewMail "${MAIL_USER_ID}"
            return $TRUE
        fi
    fi
    INDEX=`expr $INDEX + 1`
done
```

```
#
# Necipient not found in the list; add recipient to list and notify
#
MAIL_LIST_ARRAY[$INDEX]="${P_USERID}:${P_NEW_MSG_ID}:Y"
NotifyUserOfNewMail "${MAIL_USER_ID}"
return $TRUE
}
##############################################################################
# Name       : UpdateMailList
#
# Description: The function updates the mail list held in
#              $MAIL_LIST_ARRAY.
#
# Returns    : $TRUE
#
# Notes      : 1. Removes an entry for a user that's not been
#                 processed by CheckMail().
##############################################################################
UpdateMailList ( )
{
trap "HandleInterrupt " $SIGTERM # process this signal
#
# Define local variables
#
USER_PROCESSED_FLAG=""
INDEX=1
TEMP_MAIL_LIST_ARRAY[$INDEX]=""
#
# Copy the mail list into temporary array
#
while [ "${MAIL_LIST_ARRAY[$INDEX]}" != "" ]
do
    TEMP_MAIL_LIST_ARRAY[$INDEX]="${MAIL_LIST_ARRAY[$INDEX]}"
    INDEX=`expr $INDEX + 1`
done
#
# Update $MAIL_LIST_ARRAY[] with users that have been processed
# from $TEMP_MAIL_LIST_ARRAY[]
#
INDEX=1
MLI=1
while [ "${TEMP_MAIL_LIST_ARRAY[$INDEX]}" != "" ]
do
    USER_PROCESSED_FLAG=`echo ${TEMP_MAIL_LIST_ARRAY[$INDEX]} |
    ➤  cut -d':' -f3`
    if [ "${USER_PROCESSED_FLAG}" = "Y" ]
    then
        #
        # User has been processed; update list
        #
        MAIL_LIST_ARRAY[$MLI]="${TEMP_MAIL_LIST_ARRAY[$INDEX]}"
```

```
        MLI=`expr $MLI + 1`
    fi
    INDEX=`expr $INDEX + 1`
done
return $TRUE
}
##############################################################################
# Name       : CheckMail
#
# Description: Retrieves the last mail id for each recipient and
#              invokes ProcessMail().
#
# Returns    : $TRUE or $FALSE
#
# Notes      : 1. The function calls the following functions:
#                 o ProcessMail.
##############################################################################
CheckMail( )
{
trap "HandleInterrupt " $SIGTERM # process this signal
#
# Define local variables
#
MAIL_FILE_NAME=""
MSG_ID=""
#
# Switch to main mail directory
#
cd ${MAIL_DIR}
#
# Get all the file names
#
ls -1 > ${TEMP_FILE}
#
# Retrieve latest mail id
#
cat $TEMP_FILE | while read MAIL_FILE_NAME
do
    MSG_ID=`tail ${MAIL_FILE_NAME} | grep "Message-Id" | awk
    ➤  {'print $2'}`
    ProcessMail "${MAIL_FILE_NAME}" "${MSG_ID}"
done
return $TRUE
}
##############################################################################
# Name       : main
#
# Description: Invokes all other functions.
#
# Notes      : 1. The function calls the following functions:
#                 o InitializeVariables
```

```
#                   o ProcessKey
#                   o InitializeMailList
#                   o CheckMail
#                   o UpdateMailList
#
#               2. $TIME_INTERVAL is exported from schkmd.sh.
##############################################################################
main ( )
{
InitializeVariables
# trap : $SIGTERM                    # pass this signal to child
# trap "HandleInterrupt " $SIGINT    # process this signal
trap "HandleInterrupt " $SIGTERM     # process this signal
if ! ProcessKey
then
    clear
    echo "chkmd.sh:ERROR:Use schkmd.sh program to start daemon"
    exit $FEC
fi
InitializeMailList
while true
do
    CheckMail
    UpdateMailList
    sleep ${TIME_INTERVAL}
done
}
#
# invoke main
#
main
```

LISTING OF SCHKMD.SH

```
##############################################################################
# Name       : schkmd.sh (start check mail daemon)
#
# Description: The script starts the daemon script that checks mail.
#
# Input      : 1. Time interval (in seconds, default = 60 seconds).
#
# Notes      : 1. The script is run from the command line as follows:
#                 schkmd.sh t=<interval time for checking new mail>
#
#              2. It contains the following functions:
#                 o main
#                 o InitializeVariables
#                 o InitializeLogFile
#                 o InstanceCheck
#                 o PraseCommandLine
```

```
#                  o ValidateCommandLineArgumentValues
#                  o StartDaemonProcess
#                  o RootUser
#                  o DisplayMessage
#                  o WriteLog
#                  o MoveCursor
#                  o HandleInterrupt
#                  o ProcessExit
#
#              3. It starts chkmd.sh as a background task and exits.
#
#              4. Log file chkmd.log is maintained in /tmp, if the
#                 file doesn't exist, it is created.
#
#              5. If the time parameter is not supplied, a default
#                 is used.
##############################################################################
# Name       : InitializeVariables
#
# Description: Initializes all the required variables
##############################################################################
InitializeVariables ()
{
KEY="$$"; export KEY
INDEX=1                                        # working index
TIME_INTERVAL=""; export TIME_INTERVAL         # time interval
DEFAULT_TIME_INTERVAL=60                       # default is 60 seconds
TEMP_FILE="/tmp/chkmd.tmp"                     # temporary file
LOG_FILE="/tmp/chkmd.log"; export LOG_FILE     # log file name
# chkmd.sh process id file
CHKMD_PROCID_FILE="/tmp/chkmd.pid"  ;  export CHKMD_PROCID_FILE
# define message prefixes
ERROR="schkmd.sh:ERROR:"
INFO="schkmd.sh:INFO:"
# define escape sequences
ESC="\0033["
RVON=_[7m                  # reverse video on
RVOFF=_[27m                # reverse video off
BOLDON=_[1m                # bold on
BOLDOFF=_[22m              # bold off
BON=_[5m                   # blinking on
BOFF=_[25m                 # blinking off
# define return codes
TRUE=0
FALSE=1
# define exit codes
FEC=1
SEC=0
# define signal
SIGHUP=1     ; export SIGHUP
SIGINT=2     ; export SIGINT
```

```
SIGQUIT=3    ; export SIGQUIT
SIGTERM=15   ; export SIGTERM
SIGSTOP=17   ; export SIGSTOP
SIGTSTP=18   ; export SIGTSTP
# define messages
INTERRUPT="Process interrupted\; Quitting early${RVOFF}"
INVALID_ARGC="Wrong number of arguments${RVOFF}"
USAGE="Usage:schkmd.sh \[ t=\<time in seconds\> l=\<log file\>
➤  \]${RVOFF}"
DAEMON_NOT_STARTED="Failed to start daemon${RVOFF}"
NOT_ROOT_USER="Must execute the script from root account${RVOFF}"
DAEMON_ALREADY_RUNNING="An instance of daemon is aleady running${RVOFF}"
STARTING_DAEMON="Starting daemon chkmd.sh${RVOFF}"
DAEMON_STARTED="Successfully started daemon${RVOFF}"
INVALID_ARG_TYPE="\${ARG_TYPE}, is an invalid argument type${RVOFF}"
DUP_ARG="Multiple occurrence of argument type, \${ARG_TYPE}${RVOFF}"
TIME_NOT_NUMERIC="Time interval value must be numeric${RVOFF}"
TIME_NOT_VALID="Time interval value is not valid${RVOFF}"
NO_CHKMD_PROG="Program chkmd.sh does not exist in /usr/bin
➤  directory${RVOFF}"
CHKMD_NOT_EXECUTABLE="Program /usr/bin/chkmd.sh is not
➤  executable${RVOFF}"
NO_NEWMSG_PROG="Program newmsg does not exist in /usr/bin
➤  directory${RVOFF}"
NEWMSG_NOT_EXECUTABLE="Program /usr/bin/newmsg is not
➤  executable${RVOFF}"
SYSERROR="\${ERR_MSG}${RVOFF}"
}
################################################################################
# Name       : HandleInterrupt
#
# Description: Calls ProcessExit.
################################################################################
HandleInterrupt ()
{
DATETIME=`date "+%d/%m/%Y %H:%M:%S"`
echo "chkmd.sh:ERROR:Program terminated on ${DATETIME}" >> ${LOG_FILE}
ProcessExit $FEC
}
################################################################################
# Name       : MoveCursor
#
# Input      : Y and X coordinates
#
# Returns    : None
#
# Description: Moves the cursor to the required location (Y, X).
#
# Notes      : 1. This function must run in ksh for print to work.
################################################################################
MoveCursor  ()
```

```
{
trap "HandleInterrupt " $SIGTERM
YCOR=$1
XCOR=$2
echo "${ESC}${YCOR};${XCOR}H"
}
################################################################################
# Name       : DisplayMessage
#
# Description: Displays message.
#
# Input      : 1. Message type (E = Error, I = Informative)
#              2. Error Code as defined in DefineMessages ().
#              3. Message to be acknowledged flag
#
# Notes      : 1. The user must acknowledge the message if the
#                 acknowledgement flag is set to Y.
################################################################################
DisplayMessage ()
{
trap "HandleInterrupt " $SIGTERM
MESSAGE_TYPE=$1
MESSAGE_TEXT=`eval echo $2`
ACK_MESSAGE="$3"
if [ "${ACK_MESSAGE}" = "" ]
then
    ACK_MESSAGE="Y"
fi
clear
MoveCursor 24 1
if [ "${MESSAGE_TYPE}" = "E" ]
then
    echo "`eval echo ${RVON}${ERROR}`${MESSAGE_TEXT}\c"
else
    echo "`eval echo ${RVON}${INFO}`${MESSAGE_TEXT}\c"
fi
# let the user acknowledge the message
if [ "${ACK_MESSAGE}" = "Y" ]
then
    read DUMMY
fi
return ${TRUE}
}
################################################################################
# Name       : ProcessExit
#
# Description: Processes a graceful exit.
#
# Input      : 1. Exit code
################################################################################
ProcessExit ()
```

```
{
# assign parameters
EXIT_CODE="$1"
exit $EXIT_CODE
}
################################################################################
# Name       : WriteLog
#
# Description: Writes a message in the log file $LOG_FILE.
#
# Input      : 1. Message type
#              2. Message text
#
# Returns    : $TRUE
################################################################################
WriteLog ()
{
trap "HandleInterrupt " $SIGTERM
#
# assign parameters
#
MESSAGE_TYPE="$1"
MESSAGE_TEXT="$2"
DATETIME=`date "+%d/%m/%Y %H:%M:%S"`
#
# write log
#
if [ "${MESSAGE_TYPE}" = "E" ]
then
    echo "${ERROR}${DATETIME}:${MESSAGE_TEXT}" >> ${LOG_FILE}
else
    echo "${INFO}${DATETIME}:${MESSAGE_TEXT}" >> ${LOG_FILE}
fi
return $TRUE
}
################################################################################
# Name       : InstanceCheck
#
# Description: Checks whether any other instance of the program is
#              running.
#
# Returns    : $TRUE if no other instance is running
#              $FALSE otherwise.
################################################################################
InstanceCheck ()
{
trap "HandleInterrupt " $SIGTERM
if ps -eaf | grep "chkmd.sh" | grep -v "grep" > /dev/null 2>&1
then
# an instance is running
    return $FALSE
```

```
else
    return $TRUE
fi
}
##############################################################################
# Name       : RootUser
#
# Description: Checks whether the user is root.
#
# Returns    : TRUE if user is root
#              FALSE otherwise.
##############################################################################
RootUser ()
{
trap "HandleInterrupt " $SIGTERM
USER=`id | cut -d'(' -f2 | cut -d')' -f1`
if [ "${USER}" = "root" ]
then
    return $TRUE
else
    return $FALSE
fi
}
##############################################################################
# Name       : ParseCommandLine
#
# Description: Parses the command line parameters.
#
# Returns    : $TRUE or $FALSE
#
# Notes      : 1. The following command line parameters are expected:
#                 t=<time in seconds>
#                 l=<logfile name>
#
#              2. The following variables are assigned:
#
#                 Parameter  Variable Assigned
#                 t          TIME_INTERVAL
#                 l          LOG_FILE
##############################################################################
ParseCommandLine ()
{
trap "HandleInterrupt " $SIGTERM
#
# establish argument count
#
if [ ${ARGC} -gt 1 ]
then
    DisplayMessage E "${INVALID_ARGC}"
    DisplayMessage E "${USAGE}"
    return $FALSE
```

```
fi
#
# process arguments
#
INDEX=1
while [ ! $INDEX -gt $ARGC ]
do
    #
    # extract next argument line
    #
    ARG_LINE=`echo "${ARGV}" | cut -d' ' -f${INDEX}`
    #
    # extract argument type
    #
    ARG_TYPE=`echo ${ARG_LINE} | cut -c1-2`
    case "${ARG_TYPE}" in
        t= ) #
             # check for duplicate argument type
             #
             if [ "${ARG_TYPE}" = "t=" -a "${TIME_INTERVAL}" != "" ]
             then
                  DisplayMessage E "${DUP_ARG}";
             elif [ "${ARG_TYPE}" = "t=" -a "${TIME_INTERVAL}" = "" ]
             then
                 #
                 # store this argument value
                 #
                 TIME_INTERVAL=`echo "${ARG_LINE}" | cut -d'=' -f2`;
             fi;;
        *  ) DisplayMessage E "${INVALID_ARG_TYPE}";
             DisplayMessage I "${USAGE}";
             return $FALSE;;
    esac
    INDEX=`expr $INDEX + 1`
done
return $TRUE
}
##############################################################################
# Name       : ValidateCommandLineArgumentValues
#
# Description: Validates command line arguments.
#
# Returns    : $TRUE or
#              $FALSE.
##############################################################################
ValidateCommandLineArgumentValues  ()
{
trap "HandleInterrupt " $SIGTERM
#
# check whether time interval is null
#
```

```
if [ "${TIME_INTERVAL}" = "" ]
then
    TIME_INTERVAL = "${DEFAULT_TIME_INTERVAL}"
fi
#
# check that it is numeric
#
if ! [ `expr ${TIME_INTERVAL} + 1 2> /dev/null` ]
then
    DisplayMessage E "${TIME_NOT_NUMERIC}"
    return $FALSE
fi
#
# check for greater than zero
#
if ! [ ${TIME_INTERVAL} -gt 0 ]
then
    DisplayMessage E "${TIME_NOT_VALID}"
    return $FALSE
fi
return $TRUE
}
##############################################################################
# Name       : InitializeLogFile
#
# Description: Initializes the log file.
#
# Returns    : $TRUE or
#              $FALSE.
##############################################################################
InitializeLogFile ()
{
trap "HandleInterrupt " $SIGTERM
#
# does the log file exists
#
if [ -s ${LOG_FILE} ]
then
    :     # do not initialize it
else
    echo "       Log File for Check Mail Daemon" > ${LOG_FILE}
    echo "       ==============================" >> ${LOG_FILE}
fi
return $TRUE
}
##############################################################################
# Name       : StartDaemonProcess
#
# Description: Starts the script chkmd.sh at the background.
#
# Returns    : $TRUE or
```

```
#               $FALSE
################################################################################
StartDaemonProcess ()
{
trap "HandleInterrupt " $SIGTERM
#
# check the deamon program
#
if [ ! -s /usr/bin/chkmd.sh ]
then
    DisplayMessage E "${NO_CHKMD_PROG}"
    return $FALSE
elif [ ! -x /usr/bin/chkmd.sh ]
then
    DisplayMessage E "${CHKMD_NOT_EXECUTABLE}"
    return $FALSE
elif [ ! -s /usr/bin/newmsg ]
then
    DisplayMessage E "${NO_NEWMSG_PROG}"
    return $FALSE
elif [ ! -x /usr/bin/newmsg ]
then
    DisplayMessage E "${NEWMSG_NOT_EXECUTABLE}"
    return $FALSE
fi
#
# start the daemon at the background
#
nohup /usr/bin/chkmd.sh & > /dev/null 2>&1
#
# store the process id for chkmd.sh
#
echo "$!" > ${CHKMD_PROCID_FILE}
CHKMD_PID=`cat ${CHKMD_PROCID_FILE}`
if ps -eaf | grep $CHKMD_PID | grep -v "grep" > /dev/null 2>&1
then
    return $TRUE
else
    return $FALSE
fi
}
################################################################################
# Name       : main
#
# Description: The function invokes all other functions.
#
# Notes      : The function calls following functions:
#              o InitializeVariables
#              o ParseComamndLine
#              o ValidateCommandLineArgumentValues
```

```
#               o InitializeLogFile
#               o InstanceCheck
#               o RootUser
#               o StartDaemonProcess
#               o DisplayMessage
#               o WriteLog
#               o ProcessExit
#################################################################################
main ()
{
InitializeVariables
if ! RootUser
then
    DisplayMessage E "${NOT_ROOT_USER}"
    ProcessExit $FEC
fi
if ! ParseCommandLine
then
     ProcessExit $FEC
fi
if ! ValidateCommandLineArgumentValues
then
    ProcessExit $FEC
fi
if ! InitializeLogFile
then
    ProcessExit $FEC
fi
if ! InstanceCheck
then
    WriteLog E "Another instance of the program is running"
    DisplayMessage E "${DAEMON_ALREADY_RUNNING}"
    ProcessExit $FEC
fi
WriteLog I "Starting Daemon"
DisplayMessage I "${STARTING_DAEMON}"
if ! StartDaemonProcess
then
    WriteLog E "Failed to start daemon program"
    DisplayMessage E "${DAEMON_NOT_STARTED}"
    ProcessExit $FEC
else
    WriteLog I "Successfully started daemon program"
    DisplayMessage I "${DAEMON_STARTED}"
fi
ProcessExit $SEC
}
#
# capture argument count and values in global variables
#
```

```
ARGC="$#"
ARGV="$@"
#
# invoke main
#
main
```

LISTING NEWMAIL.C

```
/*********************************************************************
* Name       : newmail.c ( new mail )
*
* Description: Writes a message in a window on a terminal.
*
* Notes      : 1. The program is executed by the check mail daemon,
*                 chkmd.sh, to notify the user of new mail.
*********************************************************************/
/* include curses header file */
#include <curses.h>
/*
 * module constants for window
 */
#define WINHEIGHT 6
#define WINWIDTH 40
#define WIN_XCOR 15
#define WIN_YCOR 8
/*
 * module constants for exit codes
 */
#define SEC 0
#define FEC 1
/*********************************************************************
* Name       : main
*
* Description: Main function.
*
* Notes      : 1. Exit codes are as follows:
*                 Success SEC
*                 Failure FEC
*********************************************************************/
main ( )
{
WINDOW *wptr;           /* pointer to the window structure */

int i;
/*
 * initialize the screen
 */
initscr( );
```

```
/*
 * create a new window
 */
wptr = newwin(WINHEIGHT,WINWIDTH,WIN_YCOR,WIN_XCOR);

if (wptr == (WINDOW *) NULL)
    exit (FEC);
box(wptr,0,0);
wattron (wptr, A_REVERSE);
/*
 * reverse the video for the whole window (80 by 6);
 */
for (i = 0; i < WINHEIGHT*WINWIDTH; i ++)
    waddstr(wptr," ");
/*
 * refresh the window
 */
wrefresh(wptr);
/*
 * add message to window
 */
wmove(wptr,0,0);
wmove(wptr,0,5);
waddstr(wptr,"********* NEW MAIL **********");
wmove(wptr,1,5);
wmove(wptr,2,5);
waddstr(wptr,"    You have new mail!");
wmove(wptr,4,5);
waddstr(wptr,"  Press ctrl-l to refresh screen");
/*
 * refresh window
 */
wrefresh(wptr);
/*
 * remove all the window resources
 */
endwin ( );
/*
 * exit
 */
exit (SEC);
}
```

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

Arif Zaman
DBA/Administrator
High-Tech Software (UK)

RAID and AIX

In the beginning, disk storage was all about the number of attached physical drives. These drives were attached to the system and divided into filesystems. Most small systems had only a single drive attached, though large ones would have multiple drives that were attached through one or more controllers. Regardless of the size and number of devices, all systems had two things in common: space was limited and prices were high!

This article explains the various RAID configurations that are available and their relationship to the AIX operating system. It will let you compare their characteristics and, hopefully, help you to decide the best one for your disk subsystem.

RAID BASICS

For a long time storage needs grew faster than storage device prices fell, so the demand for larger filesystems forced system administrators to get creative. The concept of RAID storage was born. The original RAID was simple: a Redundant Array of Inexpensive Disks lashed together to appear as a single large device to the host computer system. The various levels of RAID differ in such respects as the number of disks, the way data is read and written to disk, and their throughput, reliability, availability, and price.

While RAID drives were initially based on the idea that a group of smaller drives is cheaper than a single large one, this is becoming less and less true. Beyond the potential cost savings, RAID also allows aggregate disk performance to exceed by far the speed and throughput of a single disk device. Properly configured, RAID technology also tolerates individual device failure, allowing continuous operation in spite of the occasional disk failure.

Using RAID technology to build a big disk subsystem is not a simple matter. You must understand the user's needs, the application characteristics, the overall system loading, the required reliability and uptime, and the cost factors, and have the technical knowledge and

skills necessary to implement this technology. While vendors may lead you to believe that using RAID is as simple as signing a cheque and plugging in a few cables, knowledge of the advantages and disadvantages of the various RAID levels will improve the configuration and quality of your disk subsystem.

RAID 0

RAID level 0, or just plain RAID 0, is the simplest form of RAID. It combines several small disk devices to create a single large virtual disk device. This concatenation may be done in hardware by a smart disk controller or it may be implemented in software via the operating system with its disk device drivers. It's rare to find a hardware RAID 0 disk controller – most RAID 0 implementations are software-based.

In the configuration shown in Figure 1, four disk drives are combined to form a single logical drive. A single controller manages all four devices and the operating system sees a single logical drive containing four times the space of a single disk drive. Data is written to the disk devices sequentially, as denoted by the shaded regions in Figure 1. As one drive fills up, data is written to the next disk drive.

This set-up has only one advantage: increased space. The speed of this logical device is the same as that of any disk device in the RAID set, as the disk I/O occurs to only one drive at a time. If any one of the disk

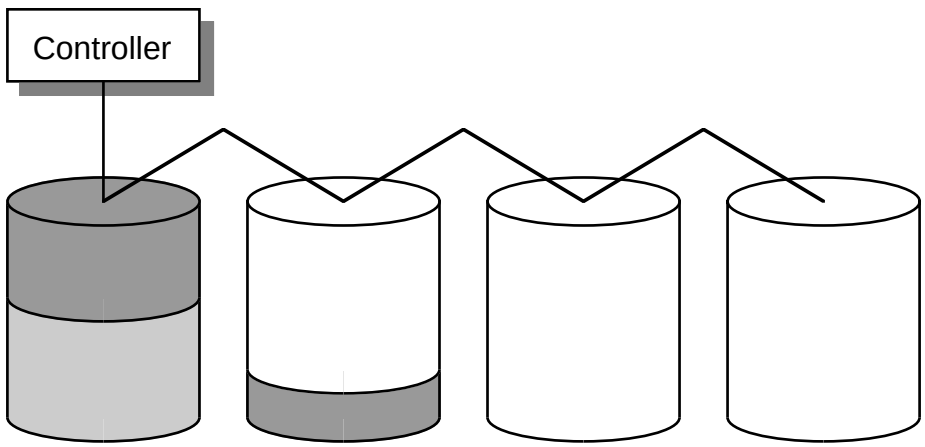

Figure 1: A simple RAID 0 configuration

devices in the RAID set fails, the whole logical volume fails. The actual reliability of the whole logical volume is four times worse than that of any one disk drive in it.

While the reliability of RAID 0 is fixed, its performance can be increased by striping. Instead of writing data sequentially to the disk drives, data can be striped across all the drives, as shown in Figure 2.

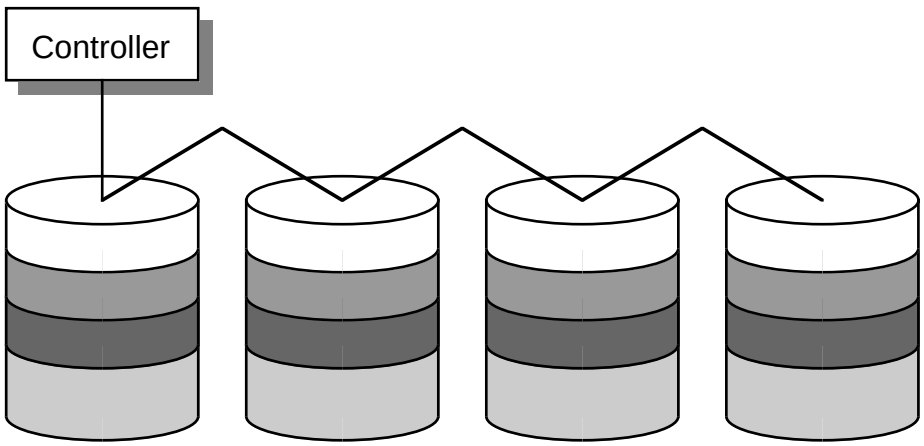

Figure 2: A striped RAID 0 configuration

In this set-up, a single I/O to the logical volume is divided into four separate operations, one for each disk drive in the RAID set. All four disk drives operate in parallel, delivering four times the data throughput to the system in the same amount of time.

Tuning the stripe size is very important – if it's too large, many I/O operations will fit in a single stripe and, hence, be restricted to a single disk drive. If the stripe is too small, too many physical operations will be initiated for each logical operation, which will saturate the controller. This is where application knowledge is a necessity. Oracle, for example, blocks all I/O into 8KB operations, which means that a four-drive RAID 0 set with a 2KB stripe size would balance each Oracle read or write across all four disk drives in the RAID set.

Striping distributes I/O evenly across the disk drives by dividing up a single I/O to the logical volume into separate disk operations. This can result in disaster at the controller if all disk drives are attached to

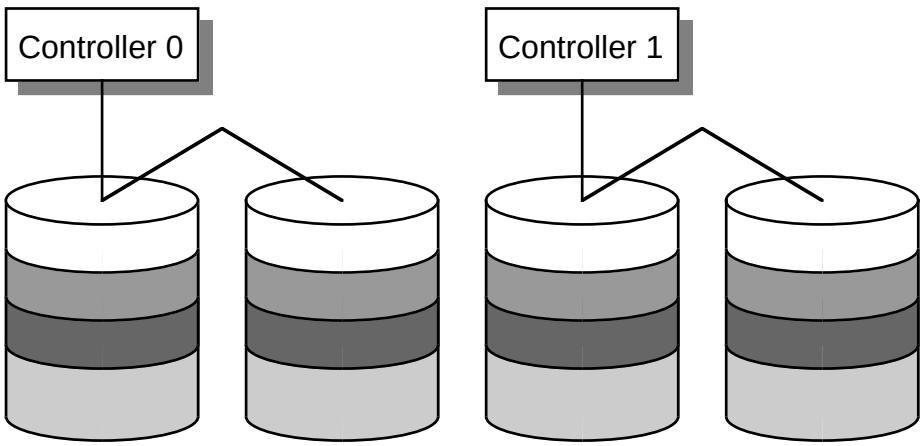

Figure 3: A striped RAID 0 with multiple controllers

a single controller. To avoid this problem, the drives should be distributed across multiple controllers, as illustrated in Figure 3.

In the example in Figure 3, I/O traffic across each controller is halved. An even better solution would be to provide each disk with its own private disk controller. In general, a SCSI controller can handle up to five disk drives. However, none of the drives in a RAID set should have to share a SCSI controller. Adding more controllers and managing I/O distribution across them can be critical when the last bit of performance needs to be wrung out of a RAID configuration.

Performance and reliability

In its simplest form, RAID 0 only offers increased disk capacity. When used in conjunction with striping, RAID 0 also provides increased performance and throughput. RAID 0 offers neither redundancy nor recovery features, and it is the least expensive form of RAID storage.

AIX and RAID 0

RAID 0 is an inherent feature of AIX's Journaled File System. As long as more than one physical volume (disk) is assigned to a logical volume, AIX supports RAID 0. If the filesystem uses a multi-disk spanned volume group, then RAID 0 is supported. Unlike other popular Unix derivatives, AIX lets you expand a filesystem by

spanning it over the logical volume. If the logical volume is too small, then it too can be expanded, even if this means installing another physical disk whose capacity is added to the logical volume. If AIX's logical volume is to be striped, then AIX 4.2.1 allows stripe sizes of 4KB, 8KB, 16KB, 32KB, 64KB, and 128KB.

RAID 1

RAID 1 is better known as mirroring. In a RAID 1 configuration, every disk device is mirrored on at least one other disk. Every write operation to a RAID 1 set results in write operations to each disk

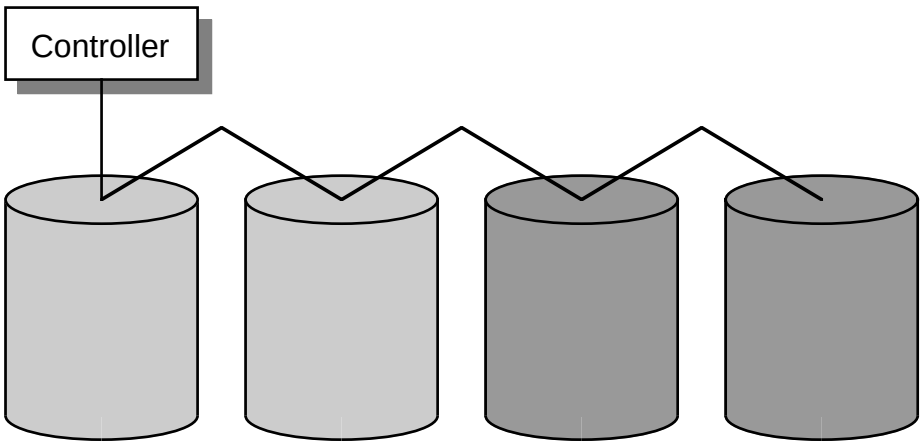

Figure 4: RAID 1 configuration with two mirrored sets

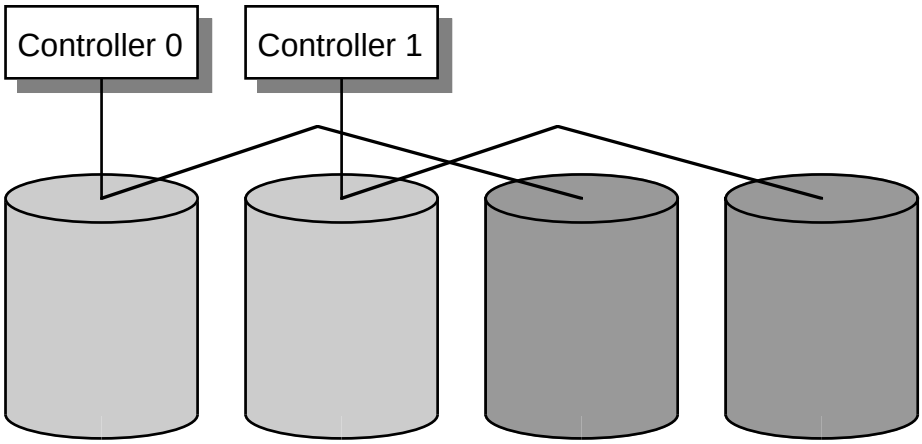

Figure 5: RAID 1 with two mirrored sets and two controllers

device in the mirror set. However, a read from a RAID 1 set results in just a single read from an available disk device. Figure 4 shows four disk drives arranged as two mirrored sets.

As a result of its nature, RAID 1 doubles the amount of physical disk capacity without increasing the amount of available virtual disk space. For this reason the use of multiple controllers is an absolute necessity for effective RAID 1 configurations – see Figure 5, which is similar to Figure 4 except that two disk controllers are used.

In this configuration, the mirrored disks use separate controllers. This increases both output and reliability as explained in the next section.

Performance and reliability

RAID 1 configurations are immune to the failure of a single disk drive. In the event of a drive failing, the system simply isolates the failed drive and transfers read and write operations to the remaining drives. A RAID 1 set running with one or more failed or missing disk drives is said to be running in 'degraded mode'. When the faulty drives are replaced, data in the remaining unmirrored drive or drives must be copied to the new drive. This operation is known as 'synching the mirror' and can take some time, especially if the affected drive is large. Again, while disk access is uninterrupted, the I/O operations needed to copy the data to the new drive steal bandwidth from user data access, possibly decreasing overall disk subsystem performance.

Using two controllers with mirrored disks increases both output and reliability. An additional controller increases output by reducing the volume of data that must pass through a single controller. Reliability is increased because, in the event of a single controller failing, the remaining controller still provides a connection to a copy of the data. Notice, however, that the data is not mirrored unless access is available to both halves of the mirrored set.

If you are committed to using RAID 1 technology to increase your system's reliability, don't be tempted to cut corners by not using enough disk controllers. I'd even go a step further and recommend using separate power supplies for each half of the mirror. Redundancy must exist at every level of the system, from drives and cables to

controllers and power supplies, before you truly have a fully mirrored disk subsystem.

AIX and RAID 1

Whether you're creating a new logical volume using the **mklv** command or making copies of an existing one with **mklvcopy**, AIX supports up to three copies of a logical volume. One copy means an unmirrored logical volume. Two or three copies mean a mirrored logical volume. You are allowed to determine which physical disks are to be used for mirroring and whether copies of a logical partition should be allocated to separate disks. You are also allowed to state which scheduling policy should be used for writing logical partitions. A serial write policy means that write procedures are carried out one after another on all the logical partitions and their copies. The write operation is not complete until all logical partitions have been written to. A parallel write policy starts the write operation on all physical partitions in a logical partition at the same time. The operation ends when the write operation to the physical partition that takes the longest to complete finishes. If the logical volume is striped, AIX 4.2.1 allows stripe sizes of 4K, 8K, 16K, 32K, 64K, and 128K.

RAID 0+1

Pure RAID 1 suffers from the same problem as pure RAID 0: data is written sequentially across the disk volumes, potentially making one drive busy while others on the side of the mirror are idle. This problem can be avoided by striping the mirrored set, just as the volumes of a RAID 0 set are striped. Theoretically, RAID 1 mirrors physical disks on a one-for-one basis. In reality, big logical disk devices are built using RAID 0 technology and then mirrored for redundancy using RAID 1 technology. This configuration is known as RAID 0+1, since it combines RAID 0's ability to aggregate capacity with RAID 1's mirroring.

Unlike RAID 0, RAID 1 and RAID 0+1 are often implemented in hardware using smart mirroring controllers. The controller manages multiple physical disk devices and presents a single logical device to the operating system.

Performance and reliability

RAID 0 and RAID 1 exist at opposite ends of the RAID spectrum. RAID 0 offers large disk volumes with no redundancy or immunity from failure, but at a low price. RAID 1 offers complete data redundancy and robust immunity from failure, but at a high cost. Both configurations can be tuned by adding controllers and using striping to distribute I/O load across as many disk drives as possible.

RAID 0+1 uses the best of both configurations, providing large volumes, high reliability, and failure immunity. However, it does not solve the price problem.

AIX and RAID 0+1

AIX 4.3.3 now supports RAID 0+1 entirely in software. This means that no special hardware is needed. AIX 4.3.3's **mklv** has a new **-s** option that specifies that no partitions from one mirror are to share disks with partitions from a second or third mirror. Another new command is **replacepv**, which allows one to replace a physical volume. If a mirror is found that is not stale, then the allocated physical partitions and their data are transferred to the destination physical volume. This enhancement is not just of interest to RAID 0+1 users, but to users of all RAID configurations.

RAID 3

RAID 3 and RAID 5 use 'parity' data to provide redundancy in the RAID volume. In simple terms, parity can be thought of as a binary checksum – a single bit of information that tells you whether all the other bits are correct. In more complex schemes, parity bits can both detect and help correct data errors.

RAID 3 takes a simple approach to using parity in a RAID configuration. Given a set of *n* drives, it uses one drive to hold parity information and stripes the data across the remaining $n - 1$ drives. Thus, in a four-drive RAID 3 set, three drives hold actual data while the fourth is dedicated to parity data. Such a configuration is often denoted as '3+1'.

Performance and reliability

The parity data contains enough information to allow data to be rebuilt if one of the data drives fails. In the previously mentioned RAID 3 '3+1' configuration, a quarter of the total number of disks is used for parity. This results in a 25% overhead, compared with RAID 1's 50%.

Reading and writing data to a RAID 3 volume can get complicated depending on the state of the data set. Reading from a healthy RAID 3 data set is not complicated. Writing to it means that, in addition to writing the actual data, the system also needs to calculate and write the parity block. Because the parity block also contains parity information from other data blocks, our write also involves at least one additional read to be able to calculate the parity block. That's quite a lot of overhead for just a simple write operation. For write-intensive applications, the parity drive cannot keep up, resulting in the RAID system slowing as a result of a parity disk bottleneck.

RAID 3 can tolerate the complete loss of a single drive but not without performance penalties. In 'degraded mode', a read from a good drive results in no additional overhead. However, reading from an unhealthy block means that reads from other blocks and parity calculations are involved before the data is retrieved.

Eventually the failed drive is replaced and the system must reconstruct it, block by block. This occurs either in the background, resulting in data traffic delays, or the RAID 3 data set is isolated from the system while the rebuild process takes place. This can be a serious concern for large RAID volumes. In simple terms, while a faulty drive can be hot swapped in just 30 seconds you may have to endure six hours of poor performance while the degraded RAID set is rebuilt.

RAID 5

The problems with the single parity drive in RAID 3 has caused almost all RAID 3 systems to shift to RAID 5 technology. RAID 5 is operationally identical to RAID 3 in that several blocks in a data stripe share a common parity block. The parity block is written whenever any data in the block is written and the parity data is used to reconstruct blocks read from a failed drive. As yet, there is no difference between the two types of RAID.

The big difference between RAID 3 and RAID 5 is that the latter distributes parity blocks throughout all the drives using an algorithm to decide where a particular stripe's parity block should reside within the drive array. Figure 6 shows the same RAID volume converted from RAID 3 to RAID 5:

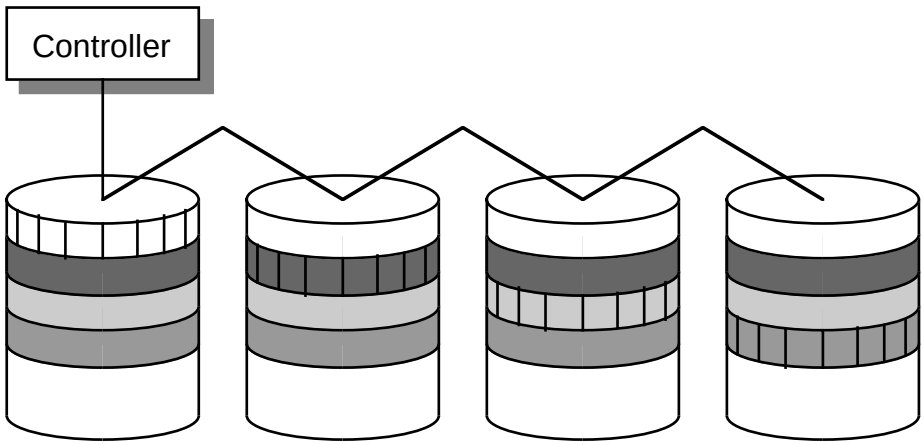

Figure 6: RAID 5

Performance and reliability

Except for eliminating the parity drive problem, RAID 5 suffers from all the same problems as RAID 3: slow writes, sensitivity to I/O block sizes, and potentially lengthy rebuilds of degraded RAID sets. In spite of these problems, RAID 5 is very popular as an economical redundant storage solution. Some of the write latency inherent with RAID 5 can be mitigated using a cached controller.

RAID 5 configurations are particularly sensitive to disk controller overloading. While the number of I/Os initiated to the RAID disk subsystem may not be enough to overload a single drive, they can often overwhelm the disk controller, which must manage all the I/O requests to all disk drives.

For this reason, it is critical that drives combined into a RAID disk subsystem are managed by separate disk controllers. That way, multiple I/Os initiated by the computer system are spread across multiple disk controllers, which then access the individual disk

drives. This prevents any one controller from being overloaded. When you are deciding which RAID 5 system to buy, pay close attention to the internal controller architecture behind the RAID controller. If all the disk drives in a RAID set are on the same device chain on a single disk controller, you could be heading for trouble.

AIX and RAID 5

AIX as an operating system does not directly support RAID 5. If the reliability of RAID 5 is needed, then hardware support in the form of the IBM 7133 SSA Subsystem, IBM 7135 RAIDiant Array, or IBM 7137 RAID Array disk subsystem needs to be used.

CONCLUSION

Now that you've been introduced to the various RAID levels and how they are directly or indirectly supported by IBM's AIX, you can hopefully design a disk subsystem that satisfies the needs of your users, system administrators, and management (which, after all, controls the money faucet!).

Werner Klauser
Klauser Informatik (Switzerland)

ZeroFault – a memory debugging tool for AIX

ZeroFault is a software tool developed by Austin, Texas-based The Kernel Group. The company has worked for a number of years with IBM's RS/6000 division and has built up considerable knowledge of the AIX operating system. This has been put to use in the design of many of the company's products, including ZeroFault.

ZeroFault's operation is based on the implementation of a Virtual Machine. The program's input (an executable program created by the user) is examined and rewritten during the course of execution. The

object code produced includes extensive tests that check each memory access. When memory-related errors are detected by ZeroFault, the location of the error and the origin of the allocated memory block are reported to the user.

The use of a Virtual Machine has many advantages that are unique to this type of application debugging:

- Only instructions that are actually executed are instrumented. This contributes to significant reductions in run times and overall memory usage.
- It's possible to enable and disable the tests performed by the application during the execution of users' programs. This is very handy for detecting errors that occur late in the life-cycle of programs being tested.
- When an error is detected, the problem is reported and the erroneous instruction is restored to its original form. Error reporting is suppressed when the erroneous instruction is re-executed – this effectively prevents the reporting of duplicate errors.
- The application is able to instrument dynamically-loaded modules invoked with the *load()* system call automatically. This automatic instrumentation is very useful with complex applications that use dynamically loaded modules heavily.
- It is possible to test applications that contain no debugging information, such as 'stripped executables'. This enables the analysis of the actual software version that is delivered to the customer.
- ZeroFault does not require the recompilation or relinking of applications being tested. This is very handy for tests that are performed by the customer or another organization that does not have access to the source code.

SUPPORTED ENVIRONMENTS

ZeroFault runs on AIX versions 3.2.5, 4.1, 4.2, and 4.3. Any AIX-

compatible system is supported, provided that the application being tested is either a POWER- or PowerPC XCOFF-formatted executable. One shortcoming is that, at present, only 32-bit executables are supported.

ZeroFault has modest system requirements: 32 MB RAM, 100 MB of swap space, and 10 MB of hard disk space. The application's installation kit can be downloaded from *ftp://ftp.zerofault.com*, and a separate installation kit is available for each major version of AIX. A demo version of the product is also available from IBM's Bonus Pack CD, which is supplied with every AIX licence. ZeroFault can be installed using either the standard **smit**-based installation screens or the **zf-custom-install** script, enabling installation even by users that don't have access to the *root* password.

Two licensing modes are available: node-locked desktop licensing and floating server-based licensing, using a licence server that can be downloaded from TKG's ftp site. The documentation is distributed in HTML format and is installed in the directory */usr/lpp/ZeroFault/doc*.

The cost of a single node-locked license is (at the time of writing) US$ 4,950, with a compulsory US$ 990 annual maintenance fee.

USER INTERFACE

ZeroFault has a Motif-based GUI and is, therefore, most useful when run from a graphical workstation. Batch-oriented usage is, however, possible using the **zf_rpt** reporting tool. In order to use ZeroFault, the user must add the directory that contains its executable to his or her *PATH* shell variable. The application that requires testing is then invoked by preceding its command with the **zf** command (including any necessary flags).

The main error pane of ZeroFault's GUI displays detected errors in outline. Lines that start with an arrow are individual messages, while ones that start with a box symbol are a group of similar messages that are condensed. The default display shows errors in summary (collapsed) form, hiding details. A verbose display contains additional information, such as memory locations and a detailed traceback of the error. Other conventions and features of the GUI are as follows:

- Memory error types are displayed using acronyms that are printed in capital letters.
- Left-clicking once on a condensed message expands the group to show the individual messages in it. Another click condenses the messages back to a group. The same also works for error messages and tracebacks.
- The source code associated with an error can be viewed if the file name is displayed in the expanded traceback. Hold down the SHIFT key and left-click the file name to invoke the source code, which is displayed in the bottom pane of ZeroFault's GUI.
- Click the 'Find Leaks' button to display the application's memory leaks – ZeroFault will display a list of the allocated memory blocks that are not referenced by any pointer.
- The 'Sort' and 'Condense by' menus, along with message filtering functions, are used to change the order and appearance of error messages.

MEMORY ERRORS DETECTED

ZeroFault is able to detect and identify many errors related to memory usage. Errors with descriptions that start with the word 'Bad' indicate that the memory referenced by the program has not been allocated by the process; errors with descriptions that start with the word 'Uninitialized' indicate that the memory used by the program has not been initialized. I'll describe each error detected by ZeroFault, illustrating each with a very short program and the associated report produced by ZeroFault.

BMR (Bad Memory Read)

This message is produced when a program tries to read from a memory location that is not allocated for its use. Some possible causes of this error include trying to read beyond the end of an allocated block, reading from a memory location that was allocated and then freed, and trying to read from random locations. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
  char *p = (char *)malloc(10);

  bzero (p,10); /* Only characters 0 to 9 are initialized */
  printf("p[10]=%d\n", p[10]);
}
```

ZeroFault reports the following information:

```
BMR      ex1    main
        Read address: 0x2040a012 -> 0x2040a012
        Read len: 1
        Read traceback:    main
                main                +0x00040(64) "ex1.c":6
        Could not find any block near it
```

BMW (Bad Memory Write)

This message is produced when a program writes to a memory location that is not allocated for its use. This is a severe error that can surface as intermittent memory errors in the later stages of the program's life-cycle. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
  char *p = (char )malloc(10); /* Only 0 to 9 are allocated */

  bzero (p,10);
  p[10]='a';
}
```

ZeroFault reports the following information:

```
BMW      ex2    main
        Write addr: 0x2040a012 -> 0x2040a012
        Write len: 1
        Write traceback:    main
                main                +0x0003c(60) "ex2.c":6
        Could not find any block near it
```

UMR (Uninitialized Memory Read)

This error message is produced when a program performs a read operation using a memory location that is not initialized by either the system or the application. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
```

```
  char *p = (char *)malloc(10);
  char c;

  bzero (p,9); /* Only characters 0 to 8 are initialized */
  c=p[9]; /* p[9] is uninitialized */
}
```

ZeroFault reports the following information:

```
UMR      ex3    main
        Access address: 0x2040a011 -> 0x2040a011
        Access len: 1
        Access traceback:    main
                main                +0x00038(56) "ex3.c":7
        Access is in block at addr 0x2040a008 -> 0x2040a011
        Block len is 10
        Block allocation traceback:    malloc    main
                malloc              +0x00000(0) "malloc.c"
                main                +0x00018(24) "ex3.c":3
```

Notice that the report includes the size and traceback of an allocated memory block that is located near the uninitialized memory location.

USTKR (Uninitialized Stack Read)

This error is produced when a program tries to access an automatic variable that's located on the program's stack before the variable is explicitly initialized. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
  char a, b;
  a=b; /* Variable b is uninitialized at this point */
}
```

ZeroFault reports the following information:

```
USTKR    ex4    main
        Current sp: 0x2ff221e0
        Read address: 0x2ff22221
        Read len: 1
        Read traceback:    main
                main                +0x0000c(12) "ex4.c":4
```

UFCP (Uninitialized Function Call Parameter)

ZeroFault checks that parameters passed to both AIX system calls and many library calls have been properly allocated and initialized, reporting failures to do so. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
  char *p = (char *)malloc(10);

  bzero(p,9);
  write(1,p,10); /* Only characters 0 to 8 are initialized */
}
```

ZeroFault reports the following information:

```
UFCP     libc.a(shr.o)    0x000037d0    write    main
        Function call: kwrite
        Parameter name: buf
        Parameter address: 0x2040a008 -> 0x2040a011
        Parameter len: 10
        Traceback:        0x000037d0    write    main
                0x000037d0 <no_symbol>
                write                +0x0010c(268) "write.c"
                main                 +0x00040(64) "ex5.c":6
        Parameter is in block at 0x2040a008
        Block len is 10
        Block allocation traceback:     malloc    main
                malloc               +0x00000(0) "malloc.c"
                main                 +0x00018(24) "ex5.c":3
```

BFCP (Bad Function Call Parameter)

This error is reported when unallocated parameters are passed to system and library function calls. The following program produces this error:

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

void main(int argc, char **argv){
        char *p=NULL;
        printf("%s\n", p);
}
```

ZeroFault reports the following information:

```
BFCP     libc.a(shr.o)    strlen    _doprnt    printf    main
        Function call: strlen
        Parameter name: string
        Parameter address: 0x0 -> 0x0
        Parameter len: 1
        Traceback:        strlen    _doprnt    printf    main
                strlen               +0x00000(0) "strlen.s"
                _doprnt              +0x01638(5688) "doprnt.c"
                printf               +0x000b8(184) "printf.c"
```

```
                main                    +0x0002c(44) "ex12.c":6
        Could not find any block near it
```

BFCT (Bad Function Call Target)

This error arises when the target of a function call is a memory location in which the function places the result of its computation and the target area is either unallocated or contains allocated blocks of insufficient length. The following program produces this error:

```
#include <stdlib.h>
void main(int argc, char **argv){
  char *s = (char *)malloc(10);

  bzero (s,20); /* The length of the target is only 10 bytes */
}
```

ZeroFault reports the following information:

```
BFCT     ex7    bzero    main
        Function call: bzero
        Parameter name: target
        Parameter address: 0x2040a008 -> 0x2040a01b
        Parameter len: 20
        Traceback:    bzero    main
                bzero                   +0x00000(0) "memset.s"
                main                    +0x0002c(44) "ex7.c":5
        Write began in block at addr 0x2040a008 -> 0x2040a011
        Block len is 10
        Block allocation traceback:    malloc    main
                malloc                  +0x00000(0) "malloc.c"
                main                    +0x00018(24) "ex7.c":3
```

BFREE (Bad Free)

This error is reported when a program tries to release a memory block that is referenced by an invalid pointer. The following program produces this error:

```
#include <stdio.h>

void main(int argc, char **argv){
  FILE *f = fopen("tmp","w");

  fprintf(f, " Hello ");
  fclose(f);
  free(f); /* The FILE structure was freed by fclose() */
}
```

ZeroFault reports the following information:

```
BFREE    ex9    main
        Free of unallocated address 0xf000ca60
        Traceback:    main
                main                +0x00050(80) "ex9.c":8
        Could not find any block near access address
```

DFREE (Double Free)

This error is reported when a program tries to release a memory block that has already been released. The following program produces this error:

```
#include <stdlib.h>

void main(int argc, char **argv){
        char *s=(char *) malloc (10);
        free(s);
        free(s); /* s has already been freed */
}
```

ZeroFault reports the following information:

```
DFREE    ex10    main
        Block addr 0x2040a008 -> 0x2040a011
        Block len is 10
        Error traceback:  main
                main                +0x00034(52) "ex10.c":6
        Allocation traceback:  main
                main                +0x00018(24) "ex10.c":4
        Free traceback:   main
                main                +0x00028(40) "ex10.c":5
```

BREALL (Bad Realloc)

This error is reported when a program tries to change the size of an allocated memory block using *realloc()*, but calls the function with an illegal pointer. This error is sometimes detected by AIX and causes an 'IOT/Abort trap(coredump)' with the following message:

```
Catastrophe in realloc: invalid storage ptr
```

The following program produces this error:

```
#include <stdlib.h>

void main(int argc, char **argv) {
        char s[10], *p;
```

```
        p = (char )realloc(s,20); /* s does not point to
                                     an allocated block */
}
```

ZeroFault reports the following information:

```
BREALL    main
        Realloc of unallocated address 0x2ff22220
        Traceback:    main
                main                +0x0001c(28) "ex11.c":6
        Could not find any block near access address
```

RNULL (Null Pointer Read)

This error is reported when a program attempts to read memory using a pointer that references memory location zero. The following program produces this error:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

void main(int argc, char ** argv) {
             char *p=0;
             putchar(*p);
}
```

ZeroFault reports the following information:

```
RNULL    rnull    main
        Access address 0x00000000
        Access len 1
        Error traceback:    main
                main                +0x0003c(60)"rnull.c":7
```

WNULL (Null Pointer Write)

This error is reported when a program attempts to update a pointer that references memory location zero. This is always detected by AIX and it causes a segmentation violation error. The following program produces this error:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

void main(int argc, char ** argv) {
             char *p=0;
```

```
            *p='c';
}
```

ZeroFault reports the following information:

```
SEGV     main
        Fault address 0x00000000
        Traceback:  main
                main                +0x0001c(28)"wnull.c":7

WNULL    wnull    main
        Access address 0x00000000
        Access len 1
        Error traceback:    main
                main                +0x0001c(28)"wnull.c":7
```

UNREF_BLOCK (Unreferenced Block)

When a user's program allocates a memory block and then destroys the pointer to it without first freeing the block, a 'memory leak' occurs. Such blocks can neither be used nor freed and are known as 'unreferenced memory blocks'. The following program produces this error:

```
void f()
{
        char *p=(char *) malloc(10);
            /* the pointer to the memory block
               is lost when the function exits */
}

main(int argc, char **argv)
{
        int i;
        for (i=0;i<10;i++)
                f();
}
```

ZeroFault reports the following information (the report is for ten leaks, but only two are shown below):

```
UNREF_BLOCK unref    f    main
        Block addr 0x2040a008 -> 0x2040a011
        Block len 10
        Allocation traceback:    f    main
                f                   +0x00010(16)"unref.c":4
                main                +0x00028(40)"unref.c":11
```

WMSG (Warning Message)

Messages of this type are produced when a process attempts to define signal handlers for signals that are used by ZeroFault or to close file descriptors that are used by the product for internal communication.

SEGV (Signal SIGSEGV received)

When a *SIGSEGV* signal is received by the user's program, ZeroFault generates this error message and terminates.

FINDING MEMORY LEAKS

ZeroFault tracks every memory allocation and de-allocation performed by the process being examined. All memory manipulation carried out by standard C functions, such as **malloc**, **realloc**, and **free**, and by C++'s **new** and **delete** operators are recorded. The best way to investigate memory usage by the program you're debugging is to use the 'Snapshot' technique. I will demonstrate this technique using the following C++ program:

```
#include <iostream.h>

class           Store {
        struct store {
                void           *ptr;
                store         *next;
        };

        store          *top;
                        public:
                        Store():top(0) {
        }
        void            insert(void *ptr);
};

void Store::insert(void *ptr)
{
        store          *s = new store;
        s->next = top;
        top = s;
        s->ptr = ptr;
        return;
}

int
```

```
main(int argc, char **argv)
{
                Store store;
                char c;

                cout<<"Press Enter to continue the program\n";
                cin>>c;

                for (int i = 0; i < 10; i++) {
                        int            *f = new int;
                        store.insert(f);
                }
                cout<<"Press ENTER to terminate the program\n";
                cin>>c;
        return 0;
}
```

The program is invoked using ZeroFault; when the program stops to wait for an input, the user takes a memory snapshot by pressing the 'Snapshot' button on the GUI. A second snapshot is taken after the program receives an input but before it runs to completion. Finally, the two snapshots are compared using the GUI's 'Compare Snapshots' button. ZeroFault displays a couple of windows that show which memory allocations differ between the two snapshots. For the sample program above, the second snapshot would contain 18 allocated memory blocks containing 108 bytes that were allocated between the two snapshots. It is possible to display an expanded listing showing the size and traceback of each allocation. It is worth mentioning that the names of C++ methods and operators appear in 'demangled form', exactly as they appear in the source code. By pressing the 'Show Leaks' button, the user is able to display a list of allocated memory blocks that are unreferenced – the sample program has 19 unreferenced memory blocks containing 152 bytes; these blocks were allocated by the C++ run-time system.

COMPARISON WITH DEBUGGING MALLOC

Issue 54 of *AIX Update* contains an article of mine entitled *Debugging Malloc in AIX 4.3.3*. This describes an AIX 4.3.3 utility that has functionality similar to that of ZeroFault. Figure 1 overleaf summarizes the features of these two utilities, highlighting their similarities and differences.

Property	Debug Malloc	ZeroFault
Price	Included with AIX 4.3	$4,950 plus $990 maintenance
AIX versions supported	4.3 only	3.2.5, 4.1, 4.2, 4.3
64-bit binary support	Unclear	No
Mode of operation	Malloc library replacement	Virtual Machine code instrumen-tation
Graphical User Interface	No	Yes
Demands relinking/recompiling	No	No
Works on stripped executables and libraries	Yes	Yes
Works with any threads package	Yes	Yes
Works with third-party applications and libraries	Yes	Yes
Works with C, C++, Fortran, Pascal, Assembler, etc	Yes	Yes
Works with dynamically loaded libraries and modules	Yes	Yes
Memory leak detection	Yes	Yes
Reads or writes unallocated heap memory	Yes	Yes
Reads or writes unallocated stack and static memory	No	Yes
Reads uninitialized stack, heap, and static memory	No	Yes
Attempts to free or reallocate unallocated memory	Yes	Yes
Passing of invalid arguments to system calls and common functions	No	Yes
Automatic ‘demangling’ of C++ names	No	Yes

Figure 1: Comparison of Debug Malloc and ZeroFault

Alex Polak
System Engineer
APS (Israel)

AIX news

McAfee has announced VirusScan Version 4.5, which protects systems against Internet-borne viruses and malicious code found in e-mail, Internet downloads, ActiveX, Java Applets, and Java Scripts. The new version is integrated with ePolicy Orchestrator, enabling virus policy management and updating over the Internet.

Another improvement is that the size of update files has been reduced to just 100 KB, which (the company claims) means that updates are delivered as much as 20 times faster than using previous distribution methods, as only the newest portion of virus signature files are downloaded.

Out now, the product supports AIX, Linux, Windows, NetWare, Solaris, HP-UX, SCO OpenServer, and MacOS. The Active Virus Defense suite starts at US$30 per node for 5,000 nodes.

For further information contact:
Network Associates Inc, 3965 Freedom Circle, Santa Clara, CA 95054, USA
Tel: +1 408 988 3832
Fax: +1 408 970 9727
Web: http://www.nai.com

Network Associates, 227 Bath Road, Slough, Berkshire SL1 5PP, UK
Tel: +44 1753 217500
Fax: +44 1753 217520

* * *

IBM has released VisualAge C++ Professional for AIX Version 5.0, the latest version of its IDE for C++, which includes both a traditional makefile-based compiler and an incremental compiler. The package supports the latest ANSI 98 C++ standard, including a complete ANSI Standard Template Library, and also supports both 32-bit and 64-bit optimization.

Included in the product is the latest C compiler for AIX, with support for RS/6000 SMP and OpenMP, a new distributed debugger for local or remote debugging, and a new performance analysis tool for C and C++ applications. Other features include a set of Open Class Libraries, an integrated and configurable IDE, a set of visual tools including Visual Builder and Data Access Builder, and HTML-based on-line help.

Other enhancements include keyboard mappings for VI and Emacs within the IDE and support for multiple code stores for use with the incremental compiler.

Out now, it costs US$2,500 per seat with upgrades priced at US$1,250.

IBM also announced Content Manager, for integrating and sharing digital content. According to the company, the software handles data in any format, including XML, HTML, images, audio, and video. It provides a single open programming interface for rapid application development and scalability. It can search across a variety of content and data repositories.

Out now, it runs on AIX and NT and prices start at US$15,000 per workstation/server and US$2,000 per concurrent user.

For further details, contact your local IBM representative.