Operating Systems Lab 05

Group 3

Example A-5. *copy-cd*: Copying a data CD

\`copy-cd.sh\` is a bash script and its purpose is to copy the contents of a data CD to an ISO file (An ISO file thus contains all the same data you would transfer when copying data to CD, DVD, or Blu-ray.) and then burn that ISO file to a blank CD-R. Below is an explanation of each line:

```
#!/bin/bash
# copy-cd.sh: copying a data CD
```

- The script starts with a shebang (\`#!/bin/bash\`) to indicate that it should be interpreted using the Bash shell.

```
CDROM=/dev/cdrom                        # CD ROM device
OF=/home/bozo/projects/cdimage.iso      # output file
#      /xxxx/xxxxxxxx/                  Change to suit your system.
BLOCKSIZE=2048
# SPEED=10                              # If unspecified, uses max spd.
# DEVICE=/dev/cdrom                     older version.
DEVICE="1,0,0"
```

- There are several variable in the script:
  - \`CDROM\` : stores the path to the CD-ROM device.
  - \`OF\`: stores the path to the output ISO file where the data from the CD will be copied.
  - \`BLOCKSIZE\` specifies the block size used for the copy operation.
  - \`DEVICE\` stores information about the CD device (in a format like "bus,target,lun").

Bus: The "bus" refers to the SCSI (Small Computer System Interface) bus to which the CD device is connected. In older systems, SCSI buses were commonly used for connecting CD-ROM drives and other peripherals. The bus number indicates the physical connection point on the SCSI bus. It helps the system identify which bus to use when communicating with the CD device. In your script, "1" is used as the bus number, but this value may vary depending on your system configuration.

Target: The "target" identifies a specific SCSI device on the chosen bus. In SCSI terminology, each device on a bus is assigned a unique target ID, typically ranging from 0 to 7. The target ID allows the system to distinguish between different devices connected to the same bus. In your script, "0" is used as the target ID, but this value can also vary based on your setup.

Lun (Logical Unit Number): The "LUN" or "Logical Unit Number" is a value used to distinguish between multiple logical units (such as partitions or sub-devices) within a single SCSI device. It is typically set to 0 for most CD-ROM drives because they usually don't have multiple logical units. However, some SCSI devices, like multifunction CD writers, may have multiple LUNs. In your script, "0" is used for the LUN.

*so since there is 1,0,0 = that means that it is using bus

```
echo; echo "Insert source CD, but do *not* mount it."
echo "Press ENTER when ready. "
read ready                              # Wait for input, $ready not used.
```

- These lines displays / prints a messages to the user, instructing them to insert the source CD but not to mount it.
-The script then waits for the user to press Enter.

```
echo; echo "Copying the source CD to $OF."
echo "This may take a while. Please be patient."

dd if=$CDROM of=$OF bs=$BLOCKSIZE         # Raw device copy.
```

- These lines provide information to the user about the copying process and tells the user to be patient

These lines use the echo command to print messages to the terminal. The first echo command is followed by a semicolon ;, which separates commands on the same line. Here's what each part does:

The first echo with no arguments is used to insert a blank line in the terminal, creating some visual separation

The second echo command prints a message to the terminal. The message informs the user that the script is in the process of copying the source CD to the specified output file ($OF). $OF

: The variable $OF contains the path to the output ISO file where the data from the CD will be copied.

- The `dd` command is used to perform a raw device copy from the CD-ROM (`$CDROM`) to the output file (`$OF`) with the specified block size (`$BLOCKSIZE`).

This line uses the dd command to perform the actual copy operation. Let's break down the components of this command:

dd: This is the command used for copying data. It stands for "data duplicator" and is a versatile command for copying files and data streams.

if=$CDROM: Here, if stands for "input file." It specifies the source of the data to be copied. $CDROM is a variable that contains the path to the CD-ROM device.

of=$OF: Similarly, of stands for "output file." It specifies where the copied data should be written. $OF is a variable containing the path to the output ISO file.

bs=$BLOCKSIZE: This sets the block size for the copy operation. The $BLOCKSIZE variable contains the value 2048, which is the size in bytes for each block of data to be copied. In this case, a block size of 2048 bytes is used, which is a common block size for CD data.

```
echo; echo "Remove data CD."
echo "Insert blank CDR."
echo "Press ENTER when ready. "
read ready                          # Wait for input, $ready not used.
```

- These lines instruct the user to remove the data CD from the drive
  - Instructs to insert a blank CD-R.
  - The script waits for the user to press Enter.

```
echo "Copying $OF to CDR."

# cdrecord -v -isosize speed=$SPEED dev=$DEVICE $OF   # Old version.
wodim -v -isosize dev=$DEVICE $OF
# Uses Joerg Schilling's "cdrecord" package (see its docs).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html
# Newer Linux distros may use "wodim" rather than "cdrecord" ...
```

Wodim:

wodim is used to record data or audio Compact Discs on an Orange Book CD-Recorder or to write DVD media on a DVD-Recorder.

- These lines copy the contents of the ISO file (`$OF`) to a blank CD-R using the `wodim` command.
-The script provides some comments about the use of "cdrecord" and "wodim."
wodim -v -isosize dev=$DEVICE $OF - the core of CD burning Process

wodim: This is the command used for burning CDs and DVDs in many Linux distributions. It's a replacement for the older cdrecord command.

-v: This option stands for "verbose." It causes wodim to provide detailed information about the burning process, including progress and status updates, which are printed to the terminal.

-isosize: This option specifies that the input file ($OF, the ISO file) contains the size of the ISO image. This is used to determine the size of the data to be burned onto the CD-R.

dev=$DEVICE: This option specifies the target CD-R device to which the ISO file will be burned. The $DEVICE variable contains information about the CD device in the format "bus,target,lun," as explained earlier. The dev option tells wodim where to send the data.

$OF: This is the path to the ISO file that will be burned to the CD-R. This is the same ISO file that was created earlier in the script by copying data from the source CD.

```
echo; echo "Done copying $OF to CDR on device $CDROM."

echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.
read answer
```

- the lines inform the user that the copying process is complete and then ask if they want to erase the image file (`$OF`). The user's response is stored in the `answer` variable.

```
case "$answer" in
([yY]) rm -f $OF
    echo "$OF erased."
    ;;
*)   echo "$OF not erased.";;
esac
```

- This `case` statement checks the user's response (`$answer`) and either deletes the image file and displays a message if the response is "y" or "Y" (yes), or displays a different message if the response is anything else.

-

rm -f $OF: If the condition [yY] is met (i.e., if $answer is "y" or "Y"), this line removes (erases) the file specified by the variable $OF. The rm -f command is used to forcefully remove a file without prompting for confirmation.

echo "$OF erased.": If the file removal is successful, this line prints a message to the terminal, indicating that the file specified by $OF has been erased.

;; : This double semicolon marks the end of the code block for the [yY] condition in the case statement.

*): This is a wildcard condition that matches any value of $answer that did not match the [yY] condition. In other words, if $answer is anything other than "y" or "Y," this block of code will be executed.

echo "$OF not erased.": If the wildcard condition matches (i.e., if $answer is not "y" or "Y"), this line prints a message to the terminal, indicating that the file specified by $OF has not been erased.

;;: This double semicolon marks the end of the code block for the wildcard condition.

esac: This line marks the end of the entire case statement. It tells the shell that there are no more conditions to check, and the script continues executing from the point immediately after the esac statement.

echo

# Exercise:
# Change the above "case" statement to also accept "yes" and "Yes" as input.

exit 0

- The script ends with an empty line and a comment suggesting an exercise to modify the `case` statement to accept "yes" and "Yes" as valid input, and then it exits with a status code of 0, indicating successful execution.

Example A-14. *fifo*: Making daily backups, using named pipes

```
#!/bin/bash
```

The `#!/bin/bash` at the beginning of a script is called a shebang or hashbang. It's a special sequence of characters that tells the system which interpreter should be used to execute the script.

1. Shebang (`#!`)
   - The `#!` characters at the start of the line are known as the shebang or hashbang.
   - This combination is a convention used in Unix-like operating systems to specify the interpreter for executing the script that follows.

2. `/bin/bash`
   - Following the shebang, `/bin/bash` specifies the interpreter to be used for running the script, in this case, the Bash shell.
   - The path `/bin/bash` indicates the location of the Bash interpreter on the system.

3. Interpretation
   - When you run a script with this shebang (e.g., `./script.sh`), the system will use the Bash interpreter located at `/bin/bash` to execute the script.
   - The script's commands and logic are interpreted and executed by the Bash shell.

```
# ==> Script by James R. Van Zandt, and used here with his permission.

# ==> Comments added by author of this document.


 HERE=`uname -n`    # ==> hostname
 THERE=bilbo
 echo "starting remote backup to $THERE at `date +%r`"
 # ==> `date +%r` returns time in 12-hour format, i.e. "08:08:34 PM".
```

1. Variable Assignment - `HERE=`uname -n``:
   - `uname -n` is a command that prints the hostname of the machine.
The uname command is used to print system information about the operating system, including the system name, node name (hostname), kernel release, kernel version, machine hardware, and processor type. -n for --nodename or hostname.
   - The backticks (``) around `uname -n` execute the `uname -n` command and substitute the output (the hostname) into the variable `HERE`.
   - `HERE` is a variable that will hold the hostname of the machine where the script is being run.

2. Variable Assignment - `THERE=bilbo`:
   - Assigns the string "bilbo" to the variable `THERE`.
   - `THERE` is a variable representing the remote machine's identifier (in this case, "bilbo").

3. Print Message - `echo "starting remote backup to $THERE at `date +%r`"`:
   - `echo` is a command that prints a message to the standard output.
   - `"starting remote backup to $THERE at `date +%r`"` is the message to be printed.
   - `$THERE` is a variable expansion, where the value of `THERE` (which is "bilbo") is substituted into the message.
   - `` `date +%r` `` is a command substitution using backticks to insert the current time in 12-hour format into the message.

   - This line prints a message indicating the start of a remote backup, mentioning the target machine (`$THERE`) and the current time in 12-hour format.

```
 # make sure /pipe really is a pipe and not a plain file
 rm -rf /pipe
 mkfifo /pipe      # ==> Create a "named pipe", named "/pipe" …
```

1. rm` Command:
   - `rm` stands for "remove" and is used to remove files and directories.
   - Options used:
     - `-r` (or `--recursive`): Recursively remove directories and their contents.
     - `-f` (or `--force`): Ignore nonexistent files and arguments and never prompt for confirmation.

     - `rm -rf /pipe` forcibly and recursively removes the named pipe `/pipe`. The `-f` option ensures that it is removed without prompting for confirmation, and the `-r` option handles the removal of directories and their contents.

2. `mkfifo` Command:
   - `mkfifo` stands for "make a FIFO special file" and is used to create a named pipe.

   - `mkfifo /pipe` This command creates a named pipe named /pipe in the root directory (/) of the file system using the mkfifo command. The named pipe will be used for communication between processes.

A named pipe is a type of special file that facilitates inter-process communication by allowing processes to communicate with each other using standard read (`<`) and write (`>`) operations. In this script, it sets up a communication channel (`/pipe`) that will be used to transfer data between processes for the backup operation.
In the script provided, /pipe is a file path that serves as a named pipe—a special type of file used for inter-process communication. The name "pipe" is arbitrary and chosen for clarity and simplicity in the script.

```
 # ==> 'su xyz' runs commands as user "xyz".
 # ==> 'ssh' invokes secure shell (remote login client).
 su xyz -c "ssh $THERE \"cat > /home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
 cd /
 tar -czf - bin boot dev etc home info lib man root sbin share usr var > /pipe
 # ==> Uses named pipe, /pipe, to communicate between processes:
 # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.
```
Let's break down the commands step by step:

1. `su` Command:
   - `su` stands for "substitute user" and is used to switch to another user account.
   - `xyz` is the username to which the command is being switched.

2. `-c` Option:
   - `-c` is an option for the `su` command, which allows you to specify a command to be executed.

3. `"ssh $THERE ..."`:
   - `"ssh $THERE ..."` is a command passed to `su` using the `-c` option.
   - This command uses `ssh` to connect to a remote machine (`$THERE`), execute a command, and transfer data.

4. `ssh` Command:
   - `ssh` is a command used to connect to a remote machine over a secure shell.
   - `$THERE` is the variable representing the remote machine name ("bilbo" in this script).

5. `"cat > /home/xyz/backup/${HERE}-daily.tar.gz"`:
   - This is the command to be executed on the remote machine via SSH.
   - `cat > /home/xyz/backup/${HERE}-daily.tar.gz` reads data from standard input (coming from the pipe) and writes it to a file on the remote machine.
   - The file is created in the `/home/xyz/backup` directory with a filename based on the local hostname (`${HERE}`) and the string "-daily.tar.gz".

6. `< /pipe`
   - `< /pipe` redirects the content from the named pipe `/pipe` to the standard input of the `cat` command within the SSH session.
   - The data that was previously written to `/pipe` (by the `tar` command) is sent to the `cat` command through this redirection.

7. `&`

- `&` puts the preceding command (the SSH command and its arguments) into the background, allowing the script to continue executing without waiting for the remote backup to complete.

8. `cd /`
   - `cd /` changes the current working directory to the root directory (`/`).

9. `tar` Command:*
   - `tar` is a command used for creating or extracting tape archives.
   - `-czf - bin boot dev etc home info lib man root sbin share usr var` specifies the directories and files to be included in the archive.
   - `-c` creates a new archive, `-z` compresses the archive using gzip, `-f -` specifies that the archive is written to standard output.

10. `> /pipe`
    - `> /pipe` redirects the output of the `tar` command (the gzipped archive) to the named pipe `/pipe`.

These commands are orchestrating the creation of a backup archive, compressing it, and then using `ssh` to transfer it to a remote machine, all while utilizing a named pipe (`/pipe`) for efficient data transfer between processes.

# ==> The end result is this backs up the main directories, from / on down.

# ==> What are the advantages of a "named pipe" in this situation, as opposed to an "anonymous pipe", with |?

- Persistence: Named pipes persist even after the processes using them have finished, allowing for communication between processes that may not be running concurrently.
- Unrelated Processes: Named pipes allow unrelated processes to communicate, while anonymous pipes are typically used for communication between related or parent-child processes.
- Flexibility: Named pipes can be used for bidirectional communication, and multiple processes can read from and write to the same named pipe simultaneously. Anonymous pipes support unidirectional communication and are usually limited to one reader and one writer.
- More Complex Communication Patterns: Named pipes allow for more complex communication patterns, including many-to-many communication scenarios, which is not possible with anonymous pipes.

# ==> Will an anonymous pipe even work here?
An anonymous pipe (|) could be used in this scenario to connect the standard output of one process (e.g., tar) to the standard input of another (e.g., ssh). However, in this script, a named

pipe (/pipe) is chosen for more flexibility and to enable communication between unrelated processes (e.g., tar and ssh).

 # ==> Is it necessary to delete the pipe before exiting the script?

It's not strictly necessary to delete the named pipe before exiting the script. The operating system will clean up resources, including named pipes, once the processes using them have finished. However, it's good practice to clean up resources like named pipes to avoid clutter and potential issues.

 # ==> How could that be done?

rm /pipe

This command will remove the named pipe /pipe. Including this in the script ensures that the named pipe is deleted at the end of script execution.

 exit 0

The exit 0 command is used to exit a shell script or a program with a specific exit status code. In this case, exit 0 is used to exit the script with an exit status of 0.

- exit: The exit command is a shell command used to terminate the execution of a script or program.
- 0: The number following the exit command is the exit status code. Exit status codes are used to indicate the success or failure of a script or program. By convention, a status code of 0 typically signifies successful execution, while non-zero status codes are used to indicate errors or issues.

Example A-23. Mounting USB keychain storage devices

```bash
# !/bin/bash

# As mentioned above the path/dir and configs, this defines the variables to specify the path to the device, the mount point, and the devlabel configuration file.
# The mount point is a directory where the usb can be attached to or mounted to make its contents accessible to the OS
# The devlabel is used to provide consistent device names for removable storage devices.
SYMLINKDEV=/dev/diskonkey
MOUNTPOINT=/mnt/diskonkey
DEVLABEL=/sbin/devlabel
DEVLABELCONFIG=/etc/sysconfig/devlabel
IAM=$0

##
# Functions lifted near-verbatim from usb-mount code.
#
# ------------------------------------------------------------------------------------------------------------------- #
# This function is used to find the attached USB storage devices
# it finds in /proc/scsi/ (scsi devices related to USB in this). It finds all files that have usb-storage in them (type -f means file) and then greps for the line that says "Attached: Yes" in the file. xargs is used to execute each file found as an argument to the grep command.
function allAttachedScsiUsb {
  find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f |
  xargs grep -l 'Attached: Yes'
}

# ------------------------------------------------------------------------------------------------------------------- #
# This function is used to determine the corresponding scsi device
# It takes as parameter an scsi device path, where echo $1 outputs the input path, and awk -F  is used to split the input path into fields (using field separator option -F) either using a hyphen or a forwards slash as the field seperator. Then through n=$(NF-1), it extracts the specific device, (NF is a special var in awk representing the number of fields) and it extracts for all but the last field (NF-1).
# Then in print, substr() extracts a single character (as specified by 1) from the string "abcdefghijklmnopqrstuvwxyz" at the position specified by n+1, where n is the extracted device number. This is then concatenated with /dev/sd to give the scsi device path.
function scsiDevFromScsiUsb {
  echo $1 | awk -F"[-/]" '{ n=$(NF-1);
  print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz", n+1, 1) }'
}

# ------------------------------------------------------------------------------------------------------------------- #
# Check if the "ACTION" env var is set to add, and if the device exists. These vars are used to determine the action to be taken and the device being acted upon - typically set by the system when the USB device is attached
# Action contains info about the action that triggered the script. Whether USB added or removed
# DEVICE represents path of the device that triggered the script
if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # lifted from usbcam code.
    #

    # --------------------------------------------------------------------------------------------------------- #
    # This block checks for the existence of console lock files (/var/run/console.lock and /var/lock/console.lock) and assigns the content of the lock file to the CONSOLEOWNER variable. These lock files are used to determine the owner of the console.
    # Console lock files indicate which user has control over the system console
    # Lock files are used to control access to resources or prevent conflicts in multi-user or multi-process environments
    if [ -f /var/run/console.lock ]; then #Check if a file name console.lock exists or not (-f to check if it is a file)
        CONSOLEOWNER=`cat /var/run/console.lock` #then contents are read using cat command and assigned to CONSOLEOWNER variable (typically the username of the user who has control over the console)
    elif [ -f /var/lock/console.lock ]; then #Check in /var/lock/ if a file name console.lock exists or not (-f to check if it is a file) - basically up one level
        CONSOLEOWNER=`cat /var/lock/console.lock` #then contents are read using cat command and assigned to CONSOLEOWNER variable (typically the username of the user who has control over the console)
    else #if neither happens, then console owner is set to an emtpty value - no specific console owner
        CONSOLEOWNER=
    fi

    # We loop over each attached USB storage device and mount it.
    for procEntry in $(allAttachedScsiUsb); do
        scsiDev=$(scsiDevFromScsiUsb $procEntry) # calls the scsiDevFromScsiUsb function to convert a SCSI device entry into a corresponding /dev/sdX device path for each entry.

        #  Some bug with usb-storage?
        #  Partitions are not in /proc/partitions until they are accessed
        #+ somehow.
        /sbin/fdisk -l $scsiDev >/dev/null  #Checks the partition information of scsi using fdisk to ensure it is recognized. The >/dev/null part discards the standard output, so only error messages (if any) are displayed.

        ##
        #  Most devices have partitioning info, so the data would be on
        #+ /dev/sd?1. However, some stupider ones don't have any partitioning
        #+ and use the entire device for data storage. This tries to
        #+ guess semi-intelligently if we have a /dev/sd?1 and if not, then
        #+ it uses the entire device and hopes for the better.
        #
        # Checks whether device has partitions or not
        # -q option stands for "quiet" and tells grep to operate silently (i.e. do not display any output).
Then append number 1 to the end of the variable,
        if grep -q `basename $scsiDev`1 /proc/partitions; then
            part="$scsiDev""1"
        else
            part=$scsiDev
        Fi

        ##
        #  Change ownership of the partition to the console user so they can
        #+ mount it.
        #
        # Check whether the COnsoleowner variable is empty or not. If not, then change the ownership
of the partition to the console user so they can mount it.
        # -z is a test operator used to check if a given string is empty
        if [ ! -z "$CONSOLEOWNER" ]; then
            chown $CONSOLEOWNER:disk $part
        fi
        ##
        # This checks if we already have this UUID defined with devlabel.
        # If not, it then adds the device to the list.
        #

        # printid is used to print the label id for a given device. -d is used to specify the device for which
we want to retrieve the label id
        prodid=`$DEVLABEL printid -d $part`
        if ! grep -q $prodid $DEVLABELCONFIG; then #if this is false, -q stands for quiet and tells grep
to operate silently (i.e. do not display any output)
            # cross our fingers and hope it works
            $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null #add the device label using 'add' to
the device through -d from the source using option -s
        Fi

        #
        # Check if the mount point exists and create if it doesn't.
        #
        # Checks if the mountpoint directory exists or not. -e is used to check if a file or dir exists.
        if [ ! -e $MOUNTPOINT ]; then
            mkdir -p $MOUNTPOINT #if not exists, then make the directory, -p is used to make parent
directories as needed - so entire directory path
        fi
```

```
        ##
        # Take care of /etc/fstab so mounting is easy.
        #

        if ! grep -q "^$SYMLINKDEV" /etc/fstab; then #Check if found or not, -q for quiet mode for grep
            # Add an fstab entry
            #here -e is used to enable interpretation of backslash escapes, in this case to format the
string output
            echo -e \
                "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0" \
                >> /etc/fstab
                # Append the echoed string to the end of the fstab file
        fi
    done

    if [ ! -z "$REMOVER" ]; then #CHeck if REMOVER variable is empty or not using -z which is a test
operator used to check if a given string is empty
        ##
        # Make sure this script is triggered on device removal.
        #
        mkdir -p `dirname $REMOVER` #if it is not empty, then make the directory including the parent
directories as needed using -p. dirname is used to extract the directory portion of the path stored in
REMOVER
        ln -s $IAM $REMOVER #ln is used to create links between files and directories , -s specifies
that a symbolic link (soft link) is to be created
    fi

elif [ "${ACTION}" = "remove" ]; then
    ##
    # If the device is mounted, unmount it cleanly.
    #
    if grep -q "$MOUNTPOINT" /etc/mtab; then
        # unmount cleanly
        # use unmount to unmount the filesystems. -l specifies lazy unmount, which allows unmounting
even if device is busy
        umount -l $MOUNTPOINT
    Fi

    ##
    # Remove it from /etc/fstab if it's there.
    #
    if grep -q "^$SYMLINKDEV" /etc/fstab; then
        # if symbolic link device is found in fstab, then remove it
        # -v is used to invert the matching behaviour of grep, it selects lines that do not match the
pattern, and are included in the output.
        grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
        # this command forcefully (-f option) renames the files, so that the new file name is the same as
the old file name.
        mv -f /etc/.fstab.new /etc/fstab
    Fi
fi

exit 0
```