Directory Organizer in SHELL

Himanshu Gangwal

June 14, 2023

Abstract

This is the documentation for the directory organizer, our project, written entirely in bash. This document explains its features and source.

Contents

1 Introduction

The Bash script presented offers a versatile solution for file organization tasks. It facilitates the seamless transfer of files from a source directory to a destination directory while providing options for customization. The script primarily categorizes files based on their extensions or creation dates, ensuring a systematic arrangement. Additionally, it grants users the ability to exclude specific file extensions, allowing for more selective organization. Moreover, the script includes the option to delete files from the source directory after successful transfer, streamlining the cleanup process. To aid in tracking and monitoring file transfers, the script also incorporates a logging feature that records the details of each transfer operation, ensuring a comprehensive record of all actions performed.
The script's functionality extends beyond regular files and encompasses zip files as well. It automatically detects and unpacks zip files recursively, ensuring that the content within them is appropriately organized. This feature adds an extra layer of convenience, eliminating the need for manual extraction.

2 Some Useful Info

The code provided is commented wherever required and this documentation will further explain the code briefly. A copy of the code is also availible here[1]. The code ha salready been dissected into sections by hashes, wherever required to further simplify the explanation. For example,

```
txtwht='\e[0;37m' # White
#########################3

##############################|
paplay $music_file & >/dev/null
music_pid=`echo "$!"` >/dev/null
##############################|

#var_management
from_dir="$1"
```

[1]https://github.com/hotramen-hellfire

2.1 A Cool Dependency !

As seen in the above snippet the code uses *paplay*, as you might have guessed the script plays a background track every time you run it. The audio file played should be in the same directory as the script with the name *bgm.mp3*. After the organization process completes the user is prompted with the option to continue or quit the audio track.

3 The CODE

The code is explained partitioned on the basis of functionality.
Let's Start,

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

3.1 Color Management

The script uses color variables which are mainly used by the echo statements to highlight the stdout and log whenever required, besides these it also makes the program look a bit cooler![2]
Snippet :

```
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
```

These are used as echo attributes as shown to set the echo output stream color,

```
echo -e "${txtgrn}FileName      SourceDir
DestinationDir      TimeStamp${txtwht}" >> $log_name
```

3.2 The BGM

As explained in 2.1 above, this piece of text is responsible to sopoth your years while your files are being organized,

[2] https://stackoverflow.com/questions/13054973/colorizing-text

Snippet :

```
paplay $music_file & >/dev/null
music_pid=`echo "$!"` >/dev/null
```

The program also prompts to stop the music before exiting, this is accomplished by storing the PID of the *paplay* process in the variable $music_pid$ and then kill the process with this PID if required at the end.

3.3 Variable Management

The `#var_management` portion is used to keep track of the variables used in an organized way.

```
from_dir="$1"
to_dir="$2"
shift
shift
del_flag=0
folders_made=0
s_chosen="ext"
exclusions=""
log_name="log.txt"
sedex='/\/[^/.]\+\.\?[^/.]\+$/p'
g_invoked=0
music_file="bgm.mp3"
```

The following variables are used in the script:

- `from_dir`: Source directory path.
- `to_dir`: Destination directory path.
- `del_flag`: Flag to indicate whether files should be deleted after transfer (0 or 1).
- `folders_made`: Number of folders created.
- `s_chosen`: Sorting option chosen ("ext" or "date").
- `exclusions`: Extensions to be excluded from sorting.

- `log_name`: Name of the log file (currently set to the default value "log.txt").
- `sedex`: Regular expression pattern used by `sed` command. (can be set by -g option)
- `g_invoked`: Flag to indicate whether the script has been invoked (0 or 1).
- `music_file`: Name of the background music file (currently set to "bgm.mp3", can be set by -m option).

3.4 GetOpts

The options are handled using the `getopts` feature in shell. The `':ds:e:l:g:m:'` specifies the options that can be used with the script, each having its own functionality. Options that require arguments are followed by a colon.

```
while getopts ':ds:e:l:g:m:' OPTION;
do
    case "$OPTION" in
        d) del_flag=1 ;;
        s) s_chosen=$OPTARG;
            if [[ ! "$s_chosen" = "ext" && ! "$s_chosen" = "date" ]];
            then
                echo "Wrong argument passed for -s, printing usage..."
                cat usage;
                kill -9 $music_pid > /dev/null;
                exit 1;
            fi ;;
        e) exclusions=$OPTARG
            flag=$(echo $exclusions | grep -c "^-")
            if [ $flag -ge 1 ]
            then
                cat usage;
                kill -9 $music_pid > /dev/null;
                exit 1;
            fi ;;
        l) log_name=$OPTARG ;
```

```
                flag=$(printf "%s" "$log_name" | grep -c '^-' )
                if [ $flag -ge 1 ]
                then
                    cat usage;
                    kill -9 $music_pid > /dev/null;
                    exit 1;
                fi ;;
            g) sedex=$OPTARG ;
                g_invoked=1
                echo "Using customized sed for extension filter (unstable)";
                flag=$(printf "%s" "$sedex" | grep -c '^-' )
                if [ $flag -ge 1 ]
                then
                    cat usage;
                    kill -9 $music_pid > /dev/null;
                    exit 1;
                fi ;;
            m) music_file=$OPTARG
                kill -9 $music_pid 2> /dev/null
                paplay $music_file & > /dev/null
                music_pid=$(echo "$!")
                flag=$(printf "%s" "$music_file" | grep -c '^-' )
                if [ $flag -ge 1 ]
                then
                    cat usage;
                    kill -9 $music_pid > /dev/null;
                    exit 1;
                fi ;;
            :) echo -e "Wrong usage, printing manual..."
                cat usage ;
                kill -9 $music_pid > /dev/null;
                exit 1 ;;
            ?) echo -e "Wrong usage, printing manual..."
                cat usage ;
                kill -9 $music_pid > /dev/null;
                exit 1 ;;
    esac
done
```

The following options are available:

- `-d`: Sets the `del_flag` variable to 1.
- `-s`: Sets the `s_chosen` variable to the value specified after `-s`. If an invalid value is provided, an error message is displayed.
- `-e`: Sets the `exclusions` variable to the value specified after `-e`. If the value starts with a dash (-), an error message is displayed.
- `-l`: Sets the `log_name` variable to the value specified after `-l`. If the value starts with a dash (-), an error message is displayed.
- `-g`: Sets the `sedex` variable to the value specified after `-g`. The `g_invoked` variable is set to 1. If the value starts with a dash (-), an error message is displayed.
- `-m`: Sets the `music_file` variable to the value specified after `-m`. It kills the previous music process, plays the new music file, and captures the process ID. If the value starts with a dash (-), an error message is displayed.
- `:` If an option requires an argument but no argument is provided, the script displays a wrong usage message and the content of the `usage` file. It then kills the music process (if any) and exits with status 1.
- `?` If an invalid option is provided, the script displays a wrong usage message and the content of the `usage` file. It then kills the music process (if any) and exits with status 1.

3.5 Zip Preprocessing

The following piece of code incorporates the functionality to recursively organize the contents of zip files.
Snippet :

```
#running zip preprocessor_ to construct recursive zip tree
 while  [ `find $from_dir -name "*.zip" -type f | wc -l` -gt 0 ] ;
 do
     for i in `find $from_dir -name "*.zip" -type f`;
```

```
    do
        name=`echo $i | awk 'BEGIN{FS="/"} {print $NF}'`
        path=`echo $i | sed -n -e "s/$name$//p"`
        unzip -o -q $i -d $path/$name'(unpack)' 2> /dev/null
        mv $i $i"^"
    done
done

for i in `find $from_dir -name "*.zip^" -type f` ;
do
   mv $i  `echo $i | sed 's/\^//g'`
done
```

The above code creates a recursive zip tree, of the form s.t. any zip file day `hello.zip` is processed to `hello(unpack).zip` along side the original zip file, as seen in the while loop condition, this process repeats itself in an iterative manner repeats until all zip files are processed.
In short :

1. The code runs in a loop as long as there are more than one `.zip` files found in the `from_dir` directory.
2. Inside the loop, it iterates over each `.zip` file found in the `from_dir` directory.
3. It extracts the file name (`name`) and the path (`path`) of the current `.zip` file using `awk` and `sed` commands.
4. The `unzip` command is used to extract the contents of the current `.zip` file into a directory named *path*/`name'(unpack)'`. The `-o` option overwrites existing files, and the `-q` option suppresses output. Any error messages are redirected to `/dev/null`.
5. After extracting the contents, the original `.zip` file is renamed by appending a caret (`^`) at the end of the filename using the `mv` command.
6. Once all the `.zip` files have been processed, another loop is used to find the renamed `.zip^` files.

7. Each renamed `.zip^` file is then renamed back to its original name by removing the caret (^) using the `sed` command.

A piece of code at the exit will recreate the original directory structure. (refer 3.11.1)

3.6 Exclusions

The code provides the option to exclude certain file extensions from sorting. In usage the argument passed to `-e` are assumed to be comma separated.
Snippet :

```
#exclusions
if [ ! $exclusions = ""  ]
then
    echo -e "t0rvalds : exclusions are on, excluding files with extensions "
    echo $exclusions | sed 's/,/ /g' > .exclusions_list
    for exc in `cat .exclusions_list 2>/dev/null`
    do
        echo -e "${txtred}$exc${txtwht}"
    done
fi
```

- A message is displayed to indicate that exclusions are enabled, informing the user that files with certain extensions will be excluded from sorting.
- The exclusions provided by the user are stored in a temporary file named `.exclusions_list`. The `echo` command is used to output the exclusions variable, and the `sed` command replaces commas (,) with spaces ( ) to separate the extensions.
- A `for` loop iterates over each exclusion in the `.exclusions_list` file.
  - The exclusion is displayed to the user, highlighted in red color using the `txtred` variable (assuming appropriate color definitions are in place).
- If the `.exclusions_list` file does not exist (indicating no exclusions were provided), any error messages are redirected to `/dev/null`.

Further more a check in the main loop is used to exclude the specified extensions.

3.7 Organizing by Extension

The code includes functionality to organize files based on their extensions. It performs the following steps:

1. If the sorting option chosen is "ext" (by extension), the code proceeds with organizing files by their extensions.

2. A message is displayed to indicate that the sorting is based on extension.
   snippet :

   ```
   if [ $s_chosen = "ext" ]
   then
       echo -e "t0rvalds : organizing by extension"
   ```

3. The code searches for all regular files (excluding directories) in the `from_dir` directory using the `find` command. Only files with an extension are selected using the `sed` command with a regular expression pattern.(can be changed using -g option)
   snippet :

   ```
   ...
   find $from_dir -type f | sed -n "$sedex"  > .operate
   if [ $g_invoked -eq 0 ];
   then
       find $from_dir -type f | sed -n '/\/[^./]$/p' >> .operate
   fi
   ...
   `for i in `cat .operate`
   ```

4. For each file (`i`), the following actions are performed:

   - The variable `count` is set to the number of times the string "(unpack)" appears in the file name, indicating whether the file is within an archive.

- The file name (`name`) and extension (`ext`) are extracted using `awk`.
- The file path (`path`) is obtained by removing the file name from the full file path using `sed`.
  snippet :

```
count=`echo $i | grep -c "(unpack)"`
name=`echo $i | awk 'BEGIN{FS="/"} {print $NF}'`
ext=`echo $name | awk 'BEGIN{FS="."} {print $NF}'`
path=`echo $i | sed -n -e "s/$name$//p"`
```

- If the file is within an archive (count is greater than or equal to 1), a message is displayed indicating the current archive path.
- If the destination directory (`to_dir`) does not exist, a message is displayed and a new directory is created using `mkdir`.
  snippet :

```
if [ ! -d $to_dir ]
then
    echo -e "${txtpur}t0rvalds : the destination
    folder doesn't exists, making one...${txtwht}"
    mkdir -p $to_dir
fi
```

- The code handles exclusions by iterating over each exclusion (`exc`) from the `.exclusions_list` file.
  - If the exclusion matches the file extension, a message is displayed indicating the exclusion and the `exclude_flag` is set to 1.
- If the file is not excluded (`exclude_flag` is not 1), the code proceeds to organize the file.
  - If a folder named `ext_`$ext^3 already exists in the destination directory, the code checks if a file with the same name exists.
  -If file with the same name already exists the name a numeric appending scheme is employed to prevent overwriting.
  snippet :

```
while [ -f $to_dir/ext_$ext/$name"."$ext ]
do
```

[3]This is the extension of the file being currently processed.

```
echo -e "t0rvalds : $name"."$ext exists in destination folder"
let "j=j+1"
name=$name_orig"_"$j
done
```

- Further a copying scheme is employed which copies the file from the source to `/tmp` then rename the file s.t. the name is unique and the finally copy it to the destination directory.
- If a file with the same name does not exist, the file is directly copied to the `ext_`*ext* directory in the destination.
- The number of transfers made is incremented, and the file path is added to the `.files_moved` file.
- Information about the file transfer is logged to the specified log file (*log_name*).

- If the `ext_`*ext* directory does not exist in the destination, a new directory is created.
  - The number of folders created is incremented.
  - The file is copied to the newly created directory.
  - The number of transfers made is incremented, and the file path is added to the `.files_moved` file.
  - Information about the file transfer is logged to the specified log file (*log_name*).
- The destination directory (`to_dir/ext_`*ext*) is added to the `.folder_list` file.

3.8 Organizing by Date

The code includes functionality to organize files based on their creation dates. It performs the following steps:

1. If the sorting option chosen is "date," the code proceeds with organizing files by their creation dates.

   Snippet:

   ```
   if [ $s_chosen = "date" ]
   then
       echo -e "t0rvalds : organizing by date created"
   ```

2. The code searches for all regular files in the specified source directory using the ‘find‘ command.

   Snippet:

   ```
   for i in `find $from_dir -type f` ;
   ```

3. For each file (`i`), the following actions are performed:

   - The code checks if the file is within an archive by searching for the string ”(unpack)” in the file name.

     Snippet:

     ```
     count=`echo $i | grep -c "(unpack)"`
     ```

   - The file name (`name`), extension (`ext`), and path (`path`) are extracted from the file’s full path.

     Snippet:

     ```
     name=`echo $i | awk 'BEGIN{FS="/"} {print $NF}'`
     ext=`echo $name | awk 'BEGIN{FS="."} {print $NF}'`
     path=`echo $i | sed -n -e "s/$name$//p"`
     ```

   - If the destination directory (`to_dir`) does not exist, a new directory is created.

     Snippet:

     ```
     if [ ! -d $to_dir ]
     then
         echo -e "t0rvalds : the destination
         folder doesn't exist,creating"
         mkdir -p $to_dir
     fi
     ```

   - The creation date of the file (`ddmmyyyy`) is obtained using the ‘stat‘ command and formatted as DDMMYYYY.

     Snippet:

     ```
     ddmmyyyy=`stat $i | sed -n '/Birth/p' |
     awk 'BEGIN{FS=" "}{print $2}' |
     awk 'BEGIN{FS="-"} {print $3$2$1}'`
     ```

   - If the file’s extension is not in the exclusion list, the code proceeds to organize the file.

     Snippet:

```
for exc in `cat .exclusions_list 2> /dev/null`
do
    if [ $exc == $ext ]
        then
        echo -e "t0rvalds : exclusion raised for
        $name as .files are excluded..."
    exclude_flag=1
    fi 2>/dev/null
done

if [ ! $exclude_flag = 1 ]
then
    # Organize the file based on the date
fi
```

- If the destination directory for the specific date does not exist, a new directory is created.

  Snippet:

  ```
  if [ ! -d $to_dir/$ddmmyyyy ]
  then
      echo -e "t0rvalds : creating directory for $ddmmyyyy"
      mkdir -p $to_dir/$ddmmyyyy
  fi
  ```

- The file is moved to the corresponding destination directory.

  Snippet:

  ```
  echo -e "t0rvalds : moving $name to $to_dir/$ddmmyyyy/"
  mv $i $to_dir/$ddmmyyyy/
  ```

The copying scheme is same as the one followed when organizing by extension as documented in section 3.7 .

3.9 Folder Statistics

The following code snippet provides information about the total number of new folders created, the total number of moves performed, and the file count for each folder in a specified folder list.

```
echo -e "Total New Folders Made : $folders_made"
echo -e "Total Moves : $((`cat $log_name | wc -l`-1))"
for i in `cat .folder_list 2>/dev/null | sort |uniq`; do
    folder_name=`echo $i | awk 'BEGIN{FS="/"} {print $NF}'`
    if [ ! $folder_name = "ext_" ]; then
        echo -e "The folder $folder_name now has `ls -A $i
        ↪  | wc -l` files."
    else
        echo -e "The folder noExtension now has `ls -A $i |
        ↪  wc -l` files."
    fi
done
```

The code includes the following steps:

1. The code displays the total number of new folders created using the variable `$folders_made`.

   ```
   echo -e "Total New Folders Made : $folders_made"
   ```

2. The code displays the total number of moves performed by subtracting 1 from the line count of the log file (`$log_name`).

   ```
   echo -e "Total Moves : $((`cat $log_name | wc
   ↪  -l`-1))"
   ```

3. The code iterates through each folder in the specified folder list (stored in the file `.folder_list`).

   ```
   for i in `cat .folder_list 2>/dev/null | sort
   ↪  |uniq`; do
       # Code block
   done
   ```

4. For each folder (`i`), the following actions are performed:

   - The folder name (`folder_name`) is extracted from the full path using the `awk` command.

```
folder_name=`echo $i | awk 'BEGIN{FS="/"} {print
↪  $NF}'`
```

- The code displays the folder name along with the count of files in the folder. It uses the `ls` command with the `-A` option (to include hidden files) piped into `wc -l` to count the files.

```
if [ ! $folder_name = "ext_" ]; then
    echo -e "The folder $folder_name now has `ls
    ↪  -A $i | wc -l` files."
else
    echo -e "The folder noExtension now has `ls -A
    ↪  $i | wc -l` files."
fi
```

This explains the purpose and functionality of the provided code, including the display of total new folders created, total moves performed, and the file count for each folder in the specified folder list.

3.10 Exit Handling

The following code snippet handles the exit process. It checks the value of the variable `del_flag` and performs specific actions accordingly.

```
echo -e "\nhandling exit : "
if [ $del_flag = 1 ]
then
    echo '-d : deleting organized files'
fi

# -d option handler
if [ $del_flag = 1 ]
then
    for i in `cat .files_moved 2>/dev/null` ;
    do
        echo -e "removing file $i..."
        rm $i
    done
fi
```

The code includes the following steps:

1. The code displays a message indicating that it is handling the exit process.

   Snippet:

   ```
   echo -e "\nhandling exit : "
   ```

2. The code checks if the value of the variable `del_flag` is equal to 1. If true, it displays a message indicating that it is deleting organized files.

   Snippet:

   ```
   if [ $del_flag = 1 ]
   then
       echo '-d : deleting organized files'
   fi
   ```

3. If the `del_flag` is equal to 1, the code enters a loop that iterates through each file listed in the file `.files_moved`. For each file (`i`), it performs the following actions:

   - The code displays a message indicating that it is removing the file.
     Snippet:
     ```
     echo -e "removing file $i..."
     ```
   - The file is removed using the `rm` command.
     Snippet:
     ```
     rm $i
     ```

   Snippet:

   ```
   if [ $del_flag = 1 ]
   then
       for i in `cat .files_moved 2>/dev/null` ;
       do
           # Code block
       done
   fi
   ```

This explains the purpose and functionality of the provided code for handling the exit process. It includes the display of messages, deletion of organized files if the `del_flag` is set to 1, and the removal of each file listed in the file `.files_moved`.

3.11 Deleting Temporaries

The following code snippet is responsible for deleting temporary files and directories.

```
# deletingTemporaries
echo -e "deleting temporary files..."

# Delete .folder_list file if it exists
if [ -f .folder_list ] ;
then
    rm .folder_list
fi

# Delete .files_moved file if it exists
if [ -f .files_moved ] ;
then
    rm .files_moved
fi

# Delete .exclusions_list file if it exists
if [ -f .exclusions_list ];
then
    rm .exclusions_list
fi

# Delete directories with name "*.zip(unpack)" if they exist
for i in `find . -type d -name "*.zip(unpack)" 2> /dev/null `;
do
    rm -rf $i 2> /dev/null
done
```

The code includes the following steps:

1. The code displays a message indicating that it is deleting temporary files.

   Snippet:

   ```
   echo -e "deleting temporary files..."
   ```

2. The code checks if the file `.folder_list` exists. If true, it removes the file using the `rm` command.

   Snippet:

   ```
   if [ -f .folder_list ] ;
   then
       rm .folder_list
   fi
   ```

3. The code checks if the file `.files_moved` exists. If true, it removes the file using the `rm` command.

   Snippet:

   ```
   if [ -f .files_moved ] ;
   then
       rm .files_moved
   fi
   ```

4. The code checks if the file `.exclusions_list` exists. If true, it removes the file using the `rm` command.

   Snippet:

   ```
   if [ -f .exclusions_list ];
   then
       rm .exclusions_list
   fi
   ```

3.11.1 Zip Postprocessing

The code searches for directories with the name "*.zip(unpack)" and deletes them. It uses the `find` command with the `-type d` option to

locate directories and the `-name` option to match the name pattern. The directories are removed using the `rm -rf` command.

Snippet:

```
for i in `find $to_dir -type d -name "*.zip(unpack)" 2> /dev/null `;
do
    rm -rf $i 2> /dev/null
done
```

This explains the purpose and functionality of the provided code for deleting temporary files and directories. It includes the deletion of specific files (`.folder_list`, `.files_moved`, `.exclusions_list`) and the removal of directories with the name pattern "*.zip(unpack)".

3.12 Stopping Music

The following code snippet allows the user to stop the music or continue vibin'.

```
# stop music?
echo -e "process completed, stop music or vibin' ? [yes/no]"
read music_flag

# Check if the user wants to stop the music
if [ $music_flag = "yes" ];
then
    kill -9 $music_pid
fi
```

The code includes the following steps:

1. The code displays a message asking the user whether they want to stop the music or continue vibin'.

   Snippet:

   ```
   echo -e "process completed, stop music or vibin' ? [yes/no]"
   read music_flag
   ```

2. The code reads the user's response and stores it in the variable `music_flag`.

   Snippet:

   ```
   read music_flag
   ```

3. The code checks if the value of `music_flag` is equal to "yes". If true, it kills the music process using the `kill` command with the `-9` option and the process ID (`$music_pid`).

   Snippet:

   ```
   if [ $music_flag = "yes" ];
   then
       kill -9 $music_pid
   fi
   ```

This explains the purpose and functionality of the provided code for stopping the music. It prompts the user to decide whether they want to stop the music or continue vibin', and if the user chooses to stop the music, it kills the associated process using the process ID stored in the variable `music_pid`.

4 Usage Brief

The Directory Organizer is a tool designed to help you organize your files in a directory. It provides several options for efficient file management. Below are the available options:

- `-d`: This option deletes the files that have been organized from the source directory. No parameter is required.
- `-e <ext1,ext2>`: Use this option to exclude specific file extensions from the organization process. Multiple extensions can be provided as a comma-separated list.
- `-l <log_name>`: Specify the name of the log file to be generated during the organization process. The log file will contain information about the files moved. Provide the desired log file name as a parameter.
- `-s <date/ext>`: Choose one of the two organization schemes: "date" or "ext." The "date" scheme organizes files based on their creation date, while the "ext" scheme organizes files by their extensions.

- `-m <param>`: Play a background music file during the organization process. By default, the file "bgm.mp3" is played. You can specify a different music file by providing its name as a parameter.
- `-g <param>`: This option takes a `sed` statement that can be used to filter files based on custom criteria. Exercise caution when using this option, as it may produce unpredictable results. By default, the `sed` statements are set to

  ```
  '/\/[^/.]\+\.\?[^/.]\+ $/p'
  ```

  and

  ```
  '/\/[^./]$/p'
  ```

  You can modify these statements to match your specific filtering needs.

Here are a few examples of using the `-g` option:

- To include only `.sh` files in your source directory:

  ```
  '/\.sh$/p'
  ```

- To include both `.sh` and `.tar` files:

  ```
  '/\.sh$\|\.tar$/p'
  ```

- To include files with regular naming such as `abc.txt` etc, use the sed :

  ```
  '/\/[^/.]\+\.[^/.]\+$\|\/[^./]$/p'
  ```

- To only get files with noextensions, use the sed :

```
'/\/[^./]\+$/p'
```

Please note that the provided `sed` statements are just examples, and you can customize them as per your requirements.

To get started, use the following syntax:

```
directory_organizer <source> <destination> [OPTIONS]
```

Example usage:

```
directory_organizer <source> <destination> -s date -l mylog.txt
```