설계과제

(1) Linux를 PC에 설치하라. 설치한 PC의 모든 주변장치를 Linux에서 전부 활용할 수 있게 튜닝할 수 있는가?

(2) Linux Box를 Web Server로 활용할 수 있게 하라. 전자 게시판도 설치하여 공지사항을 관리할 수 있게 하라.

(3) CD Collection을 관리할 Shell Script를 작성하라. (참고: 첨부자료 'Shell Script for Managing a CD Collection.pdf')

Command Line Option	set Option	Description
sh −n <script>	set −o noexec set −n	Checks for syntax errors only; doesn't execute commands.
sh −v <script>	set −o verbose set −v	Echoes commands before running them.
sh −x <script>	set −o xtrace set −x	Echoes commands after processing on the command line.
	set −o nounset set −u	Gives an error message when an undefined variable is used.

You can set the set option flags on, using −o, and off, using +o, and likewise for the abbreviated versions.

You can achieve a simple execution trace by using the xtrace option. For an initial check, you can use the command line option, but for finer debugging, you can put the xtrace flags (setting an execution trace on and off) inside the script around the problem code. The execution trace causes the shell to print each line in the script, with variables expanded, before executing the line. The level of expansion is denoted (by default) by the number of + signs at the start of each line. You can change the + to something more meaningful by setting the PS4 shell variable in your shell configuration file.

In the shell, you can also find out the program state wherever it exits by trapping the EXIT signal, with a line something like this placed at the start of the script:

```
trap 'echo Exiting: critical variable = $critical_variable' EXIT
```

Putting it All Together

Now that we've seen the main features of the shell as a programming language, it's time to write an example program to put some of what we have learned to use.

Throughout this book, we're going to be building a CD database application to show the techniques we've been learning. We start with a shell script, but pretty soon we'll do it again in C, add a database, and so on. So, let's start.

Requirements

We're going to design and implement a program for managing CDs. Suppose we have an extensive CD collection. An electronic catalogue seems an ideal project to implement as we learn about programming UNIX.

We want, at least initially, to store some basic information about each CD, such as the label, type of music and artist or composer. We would also like to store some simple track information.

We want to be able to search on any of the 'per CD' items, but not on any of the track details.

To make the mini−application complete, we would also like to be able to enter, update and delete all the information from within the application.

Design

The three requirements−updating, searching and displaying the data−suggest that a simple menu will be adequate. All the data we need to store is textual and, assuming our CD collection isn't too big, we have no need for a complex database, so some simple text files will do. Storing information in text files will keep our application simple and if our requirements change, it's almost always easier to manipulate a text file than any other sort of file. As the last resort, we could even use an editor to manually enter and delete data, rather than write a program to do it.

We need to make an important design decision about our data storage: will a single file suffice and, if so, what format should it have? Most of the information we expect to store occurs only once per CD (we'll skip lightly over the fact that some CDs contain the work of many composers or artists), except track information. Just about all CDs have more than one track.

Should we fix a limit on the number of tracks we can store per CD? That seems rather an arbitrary and unnecessary restriction, so let's reject that idea straight away!

If we allow a flexible number of tracks, we have three options:

- Use a single file, use one line for the 'title' type information and then 'n' lines for the track information for that CD.
- Put all the information for each CD on a single line, allowing the line to continue until no more track information needs to be stored.
- Separate the title information from the track information and use a different file for each.

Only the third option allows us to easily fix the format of the files, which we'll need to do if we ever wish to convert our database into a relational form (more on this in Chapter 7), so that's the option we'll choose.

The next decision is what to put in the files.

Initially, for each CD title, we'll choose to store:

- The CD catalog number
- The title
- The type (classical, rock, pop, jazz, etc.)
- The composer or artist

For the tracks, simply:

- Track number
- Track name

In order to 'join' the two files, we must relate the track information to the rest of the CD information. To do this, we'll use the CD catalog number. Since this is unique for each CD, it will appear only once in the titles file and once per track in the tracks file.

Let's look at an example titles file:

Catalog	Title	Type	Composer
CD123	Cool sax	Jazz	Bix

CD234	Classic violin	Classical	Bach
CD345	Hits99	Pop	Various

And its corresponding tracks file:

Catalog	Track No.	Title
CD123	1	Some jazz
CD123	2	More jazz
CD345	1	Dizzy
CD234	1	Sonata in D minor

The two files 'join' using the Catalog field. Remember, there are normally multiple rows in the tracks file for a single entry in the titles file.

The last thing we need to decide is how to separate the entries. Fixed−width fields are normal in a relational database, but are not always the most convenient. Another common method is a comma, which we'll use here (i.e. a comma−separated variable, or CSV, file).

In the following Try It Out, just so you don't get totally lost, we'll be using the following functions:

```
get_return()
get_confirm()
set_menu_choice()
insert_title()
insert_track()
add_record_tracks()
add_records()
find_cd()
update_cd()
count_cds()
remove_records()
list_tracks()
```

Try It Out − A CD Application

1. First in our sample script is, as always, a line ensuring that it's executed as a shell script, followed by some copyright information:

```
#!/bin/sh

# Very simple example shell script for managing a CD collection.
# Copyright (C) 1996-99 Wrox Press.

# This program is free software; you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by the
# Free Software Foundation; either version 2 of the License, or (at your
# option) any later version.

# This program is distributed in the hopes that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
# Public License for more details.

# You should have received a copy of the GNU General Public License along
# with this program; if not, write to the Free Software Foundation, Inc.
# 675 Mass Ave, Cambridge, MA 02139, USA.
```

2. The first thing to do is to ensure that some global variables that we'll be using throughout the script are set up. We set the title and track files and a temporary file. We also trap *Ctrl−C,* so our temporary file is removed if the user interrupts the script.

```
menu_choice=""
current_cd=""
title_file="title.cdb"
tracks_file="tracks.cdb"
temp_file=/tmp/cdb.$$
trap 'rm -f $temp_file' EXIT
```

3. Now we define our functions, so that the script, executing from the top line, can find all the function definitions before we attempt to call any of them for the first time.

   To avoid rewriting the same code in several places, the first two functions are simple utilities.

```
get_return() {
  echo -e "Press return \c"
  read x
  return 0
}

get_confirm() {
  echo -e "Are you sure? \c"
  while true
  do
    read x
    case "$x" in
      y | yes | Y | Yes | YES )
        return 0;;
      n | no  | N | No  | NO )
        echo
        echo "Cancelled"
        return 1;;
      *) echo "Please enter yes or no" ;;
    esac
  done
}
```

4. Here, we come to the main menu function, set_menu_choice. The contents of the menu vary dynamically, with extra options being added if a CD entry has been selected.

   Note that echo −e may not be portable to some shells.

```
set_menu_choice() {
  clear
  echo "Options :-"
  echo
  echo "   a) Add new CD"
  echo "   f) Find CD"
  echo "   c) Count the CDs and tracks in the catalog"
  if [ "$cdcatnum" != "" ]; then
    echo "   l) List tracks on $cdtitle"
    echo "   r) Remove $cdtitle"
    echo "   u) Update track information for $cdtitle"
  fi
  echo "   q) Quit"
  echo
  echo -e "Please enter choice then press return \c"
  read menu_choice
```

```
  return
}
```

5. Two more very short functions, insert_title and insert_track, for adding to the database files. Though some people hate one−liners like these, they help make other functions clearer.

   They are followed by the larger add_record_track function that uses them. This function uses pattern matching to ensure no commas are entered (since we're using commas as a field separator) and also arithmetic operations to increment the current track number as tracks are entered.

```
insert_title() {
  echo $* >> $title_file
  return
}

insert_track() {
  echo $* >> $tracks_file
  return
}

add_record_tracks() {
  echo "Enter track information for this CD"
  echo "When no more tracks enter q"
  cdtrack=1
  cdttitle=""
  while [ "$cdttitle" != "q" ]
  do
      echo -e "Track $cdtrack, track title? \c"
      read tmp
      cdttitle=${tmp%%,*}
      if [ "$tmp" != "$cdttitle" ]; then
        echo "Sorry, no commas allowed"
        continue
      fi
      if [ -n "$cdttitle" ] ; then
        if [ "$cdttitle" != "q" ]; then
          insert_track $cdcatnum,$cdtrack,$cdttitle
        fi
      else
        cdtrack=$((cdtrack-1))
      fi
    cdtrack=$((cdtrack+1))
  done
}
```

6. The add_records function allows entry of the main CD information for a new CD.

```
add_records() {
  # Prompt for the initial information

  echo -e "Enter catalog name \c"
  read tmp
  cdcatnum=${tmp%%,*}

  echo -e "Enter title \c"
  read tmp
  cdtitle=${tmp%%,*}

  echo -e "Enter type \c"
  read tmp
  cdtype=${tmp%%,*}
```

```
  echo -e "Enter artist/composer \c"
  read tmp
  cdac=${tmp%%,*}

  # Check that they want to enter the information

  echo About to add new entry
  echo "$cdcatnum $cdtitle $cdtype $cdac"

  # If confirmed then append it to the titles file

  if get_confirm ; then
    insert_title $cdcatnum,$cdtitle,$cdtype,$cdac
    add_record_tracks
  else
    remove_records
  fi

  return
}
```

7. The find_cd function searches for the catalog name text in the CD title file, using the grep command. We need to know how many times the string was found, but grep only returns a value telling us if it matched zero times or many. To get around this, we store the output in a file, which will have one line per match, then count the lines in the file.

   The word count command, wc, has whitespace in its output, separating the number of lines, words and characters in the file. We use the $(wc −l $temp_file) notation to extract the first parameter from the output to set the linesfound variable. If we wanted another, later parameter we would use the set command to set the shell's parameter variables to the command output.

   We change the IFS (Internal Field Separator) to a , (comma), so we can separate the comma−delimited fields. An alternative command is cut.

```
find_cd() {
  if [ "$1" = "n" ]; then
    asklist=n
  else
    asklist=y
  fi
  cdcatnum=""
  echo -e "Enter a string to search for in the CD titles \c"
  read searchstr
  if [ "$searchstr" = "" ]; then
    return 0
  fi

  grep "$searchstr" $title_file > $temp_file

  set $(wc -l $temp_file)
  linesfound=$1

  case "$linesfound" in
  0)    echo "Sorry, nothing found"
        get_return
        return 0
        ;;
  1)    ;;
  2)    echo "Sorry, not unique."
```

```
        echo "Found the following"
        cat $temp_file
        get_return
        return 0
    esac

    IFS=","
    read cdcatnum cdtitle cdtype cdac < $temp_file
    IFS=" "

    if [ -z "$cdcatnum" ]; then
      echo "Sorry, could not extract catalog field from $temp_file"
      get_return
      return 0
    fi

    echo
    echo Catalog number: $cdcatnum
    echo Title: $cdtitle
    echo Type: $cdtype
    echo Artist/Composer: $cdac
    echo
    get_return

    if [ "$asklist" = "y" ]; then
      echo -e "View tracks for this CD? \c"
        read x
      if [ "$x" = "y" ]; then
        echo
        list_tracks
        echo
      fi
    fi
    return 1
}
```

8. update_cd allows us to re−enter information for a CD. Notice that we search (grep) for lines that start (^) with the $cdcatnum followed by a ,, and that we need to wrap the expansion of $cdcatnum in {} so we can search for a , with no whitespace between it and the catalogue number. This function also uses {} to enclose multiple statements to be executed if get_confirm returns true.

```
update_cd() {
  if [ -z "$cdcatnum" ]; then
    echo "You must select a CD first"
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "Current tracks are :-"
    list_tracks
    echo
    echo "This will re-enter the tracks for $cdtitle"
    get_confirm && {
      grep -v "^${cdcatnum}," $tracks_file > $temp_file
      mv $temp_file $tracks_file
      echo
      add_record_tracks
    }
  fi
  return
}
```

9. count_cds gives us a quick count of the contents of our database.

```
count_cds() {
  set $(wc -l $title_file)
  num_titles=$1
  set $(wc -l $tracks_file)
  num_tracks=$1
  echo found $num_titles CDs, with a total of $num_tracks tracks
  get_return
  return
}
```

10. remove_records strips entries from the database files, using grep −v to remove all matching strings. Notice we must use a temporary file.

    If we tried to do this,

    ```
    grep -v "^$cdcatnum" > $title_file
    ```

    the $title_file would be set to empty by the > output redirection before the grep had chance to execute, so grep would read from an empty file.

    ```
    remove_records() {
      if [ -z "$cdcatnum" ]; then
        echo You must select a CD first
        find_cd n
      fi
      if [ -n "$cdcatnum" ]; then
        echo "You are about to delete $cdtitle"
        get_confirm && {
          grep -v "^${cdcatnum}," $title_file > $temp_file
          mv $temp_file $title_file
          grep -v "^${cdcatnum}," $tracks_file > $temp_file
          mv $temp_file $tracks_file
          cdcatnum=""
          echo Entry removed
        }
        get_return
      fi
      return
    }
    ```

11. List_tracks again uses grep to extract the lines we want, cut to access the fields we want and then more to provide a paginated output. If you consider how many lines of C code it would take to re−implement these 20−odd lines of code, you'll appreciate how powerful a tool the shell can be.

    ```
    list_tracks() {
      if [ "$cdcatnum" = "" ]; then
        echo no CD selected yet
        return
      else
        grep "^${cdcatnum}," $tracks_file > $temp_file
        num_tracks=$(wc -l $temp_file)
        if [ "$num_tracks" = "0" ]; then
          echo no tracks found for $cdtitle
        else {
          echo
          echo "$cdtitle :-"
          echo
          cut -f 2- -d , $temp_file
          echo
        } | ${PAGER:-more}
        fi
    ```

```
  fi
  get_return
  return
}
```

12. Now all the functions have been defined, we can enter the main routine. The first few lines simply get the files into a known state, then we call the menu function, set_menu_choice, and act on the output.

    When quit is selected, we delete the temporary file, write a message and exit with a successful completion condition.

```
rm -f $temp_file
if [ ! -f $title_file ]; then
  touch $title_file
fi
if [ ! -f $tracks_file ]; then
  touch $tracks_file
fi

# Now the application proper

clear
echo
echo
echo "Mini CD manager"
sleep 1

quit=n
while [ "$quit" != "y" ];
do
  set_menu_choice
  case "$menu_choice" in
    a) add_records;;
    r) remove_records;;
    f) find_cd y;;
    u) update_cd;;
    c) count_cds;;
    l) list_tracks;;
    b)
      echo
      more $title_file
      echo
      get_return;;
    q | Q ) quit=y;;
    *) echo "Sorry, choice not recognized";;
  esac
done

#Tidy up and leave

rm -f $temp_file
echo "Finished"
exit 0
```

Notes

The trap command at the start of the script is intended to trap the user pressing *Ctrl−C*. This may be either the EXIT or the INT signal, depending on the terminal setup.

There are other ways of implementing the menu selection, notably the select construct in bash and ksh (which, however, isn't specified in X/Open) which is a dedicated menu choice selector. Check it out if your script can afford to be slightly less portable. Multi−line information given to users could also make use of here documents.

You might have noticed that there's no validation of the primary key when a new record is started; the new code just ignores the subsequent titles with the same code, but incorporates their tracks into the first title's listing:

```
1 First CD Track 1
2 First CD Track 2
1 Another CD
2 With the same CD key
```

We'll leave this and other improvements to your imagination and creativity, as you can modify the code under the terms of the GPL.

Summary

In this chapter, we've seen that the shell is a powerful programming language in its own right. Its ability to call other programs easily and then process their output makes the shell an ideal tool for tasks involving the processing of text and files.

Next time you need a small utility program, consider whether you can solve your problem by combining some of the many UNIX commands with a shell script. You'll be surprised just how many utility programs you can write without a compiler.