The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2012

COMP3301/COMP7308
Assignment 0 — Shell using system calls

Due: N/A (not marked)

Version 1.0 — 14 May 2012

Introduction

The goal of this assignment is to refresh your knowledge of the C programming language and the skills you will need to work at the kernel-level interface of an operating system (Linux). Your task is to create a simple shell that must use system calls (rather than the higher-level C library wrappers) to access the kernel. You must also answer some short–response questions that will test your understanding of the concepts that you have implemented.

This assignment carries no weighting in the course and will not be marked, however, it is strongly recommended you complete it. The concepts and practice you will gain from this assignment will be useful in the following assignments.

Program behaviour

The program you must produce will be called `shell` and will take no command-line arguments. You do not have to handle the case where arguments are given to your program, though your program should not crash.

When started your program should wait for commands to be given (one per line, terminated by a `\n`) from *standard in*. The command should then be executed and its output printed to *standard out* (if any). Another command should then be accepted.

If `EOF` is read on *standard in*, your program should exit with status 0. This is the only time your program should exit (other than the usual signals which you do not have to handle). You may assume that each line of input will have no preceding whitespace and that each argument (if any) is separated by a single space. Blank lines should be ignored. Your program does not need to handle processing input lines longer than 1024 characters long (including newline). If it reads a line longer than this, it should print out `input was too long\n` to *standard error* and wait for another line to be input.

If a line starts with a command that is not specified below, your program should output `unrecognised command %s\n` to *standard error* (where `%s` is the name of the command).

You should ensure your program behaves correctly when *standard in* is redirected from a file.

Make sure you print any messages or output stated in this specification exactly as shown, including any newline. There should be no whitespace before or after the message.

System calls *vs.* C library functions

The *primary functionality* of the assignment must be implemented with system calls only. Miscellaneous I/O (for example user input) may use the convenience functions in `stdio.h`, `stdlib.h`, `string.h`, etc.

If you are unsure if a function is a library function or a system call, the following command may be useful:

```
man -k <foo> | grep '\(2\)'
```

(where `<foo>` is the function you are interested in). If the function you are looking for is not listed in the output, it is not a system call.

Builtin commands

You must implement three builtin commands, as specified below. You may assume that each command will be given the correct number of arguments as specified.

1: List a file or directory contents recursively

```
lsr [filename|directory]
```

The command `lsr` has two modes, depending on the argument given.

If the argument given is a filename, list the details of that file only. The following attributes should be printed in order (each separated by a space): entry type, size (in bytes) padded to 10 spaces, and the filename. The filename should have its leading directory stripped (if any). See below for an example.

The type of an entry to be printed is given in the following table:

regular file	`f`
directory	`d`
symbolic link	`s`
any other type	`o`

If the argument given is a symbolic link, details on the link should be returned — not the file it refers to.

For example, typing `lsr /bin/bash` might show the following:

```
f     934336 bash
```

If the argument given is a directory (or no argument is given, in which case assume it is the current directory), first list each entry in that directory using the format described above, then recurse into each subdirectory (if any) and list the entries in that directory (and so on). You should make sure you list all the entries in a parent directory before recursing into a subdirectory. Before printing the contents of a directory, your program should print the directory's name followed by a colon. After listing a directory's contents, a blank line should be printed. See below for an example.

A directory listing should list the files in the order returned by the kernel: no further sorting is required. You need not perform any preprocessing on paths given. The `.` and `..` entries should be ignored and not included in the output.

For example, typing `lsr` without any argument might show the following:

```
.:
f        102 Makefile
f      19123 shell
f       7212 shell.c
f      15504 shell.o
d         72 tests
s         27 mark.sh
o       5433 device

./tests:
d         72 cases
f        455 README
```

```
./tests/cases:
f          0 test-case-1
```

Your program should print a *descriptive* error to *standard error* if the filename or directory given could not be accessed (this includes any permission problems). You may wish to consider using `perror(3)` for printing errors. All other output must be written to *standard out*.

Hint: read the manual pages for the `getdents(2)` and `stat(2)` system calls in their entirety!

2: Copy a regular file

`cp source dest`

Copy the *regular file* given by the source argument to the given destination. After a successful copy, the contents of the destination file *must be identical* to the source: this command must work with both text and binary files. Text files may contain both both ASCII and Unicode characters and this should not stop your program from functioning correctly. There should be no upper limit to the size of files that your program can copy. You may assume sufficient disk space exists to copy to the destination.

Attempting to copy any other type of source file (symbolic link, directory, block special, *etc.*) should cause the error message `source file type not supported\n` to be printed and no copy should be performed.

If the destination file does not exist, permissions on the source file should be preserved (excluding ownership) when creating it, including any execute bit that was set. You do not have to handle changing permissions on the destination file if it already exists.

If the destination file exists and is a regular file, your program should attempt to overwrite it. If it is a directory, then your program should copy the source file into that directory, using the same filename as the source (for instance `cp foo subdir` will attempt to copy `foo` to `subdir/foo`). Attempting to copy to any other type of destination file should fail with the error message `destination file type not supported\n`.

If the source file cannot be opened (for whatever reason), then the destination file should not be created, or modified in any way if it already exists. Consider using `perror(3)` for providing descriptive error messages when the source file cannot be opened. Provided the copy is successful and there was no error, nothing should be printed to *standard out*. Any errors that do occur should be printed to *standard error*.

Do not attempt to copy one byte at a time — this is very inefficient.

You do not have to handle the case where the same source and destination filenames given to `cp` are the same.

Hint: the `diff` and `cmp` commands may be useful for validating that the contents of two files are identical.

3: Remove a regular file or directory recursively

`rm filename|directory`

Remove the regular filename or directory tree given from the file system. This should act in the same way as `rm(1)` does with the `-rf` arguments.

Any symbolic links encountered should cause the link to be removed, not the file they reference.

If the argument given is a directory, your program should attempt to remove the directory and its contents from the system. Your program is expected to handle any type of file or directory (including deep directory trees) that are encountered.

If the file or directory specified does not exist, or a permissions problem prevents its removal, a descriptive error should be printed to *standard error*.

Short-response questions

1. Give a brief overview of the algorithm you used to implement the recursive directory listing. Pseudocode is not required.

2. What Linux-specific privilege would be required to preserve ownership on files that are copied? Describe how one would obtain this privilege.

3. What is the practical difference between using the `getdents(2)` system call and the `scandir(3)` library call? Some things to consider might be portability, safety and ease of use.

4. Copying data using `read(2)` and `write(2)` is the conventional way to implement the copy functionality specified in this assignment. However there is another system call that is designed for this purpose and is more efficient. Which system call is this, and what are the differences between using it and the standard `read`/`write` system calls? Why is it more efficient?

Code compilation

You should provide a `Makefile` that will build your program and produce the `shell` executable when invoked by `make`; that is:

```
make
```

The `Makefile` should use `gcc` with the `-Wall -std=gnu99` compile flags. Compilation should not yield any warnings or errors.

You must not link with any other libraries other than the default C library.

Coding style

Your solution must conform to the Linux kernel coding style document available at `http://www.kernel.org/doc/Documentation/CodingStyle` (or `Documentation/CodingStyle` in the kernel source tree). You may wish to run your code through the `checkpatch` tool to validate that your solution adheres to the coding style. The following arguments may be useful to you (where `FILE` is the C source file you wish to check):

```
checkpatch --no-tree --no-signoff -f --no-summary FILE
```

The tool is available to download from the course website (under Resources).

You may also find the `indent(1)` tool useful with the following arguments:

```
indent -kr -i8 FILE
```

Please note that these are tools only — they should not be considered perfect. Always double-check the results by hand to be sure.

Submission and Version Control

This assignment has no submission as it will not be marked.

However, it is suggested that you store your assignment source code in EAIT's Subversion system (you should already be familiar with this from previous courses). You should ensure all of your source files (including any `.h` files if you have created them) and `Makefile` are committed to the repository under the following URL (where *s4123456* is your student number):

`https://svn.itee.uq.edu.au/repo/comp3301-s4123456/a0`

For information regarding ITEE's Subversion system, see `http://studenthelp.itee.uq.edu.au/faq/subversion/`.

You will be submitting future assignments using this method.