A0: Getting Started with `file(1)`

Computer Systems 2019
Department of Computer Science
University of Copenhagen

Oleks Shturmov <oleks@oleks.info>
Michael Kirkedal Thomsen <m.kirkedal@di.ku.dk>

Due: Sunday, September 08, 10:00
Version 1

This is the first in a long and winding series of assignments in the course Computer Systems 2019 at DIKU. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment does not belong to a category. Resubmission is not possible.

The purpose of this assignment is to get you started with C, and a couple command-line utilities common to a Unix-like development environment. What better way to do that, than try to implement one of these yourself?

Introduction

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*
>
> — James Whitcomb Riley, *American poet* (1849–1916)

`file(1)`[1] is a classical Unix-like command-line utility for guessing the type of a file. In a Unix-like operating system, it is often the contents, or the metadata of a file, not a filename extension, that determines the "type" of a file. Standard `file(1)` uses clever heuristics to guess the file type. Throughout A0 and A1, we will implement a basic variant of this tool in C.

In A0, we will only discern between the file types `empty`, `ASCII text`, and `data`. An `empty` file has 0 bytes. An `ASCII text` file only contains characters from a particular subset of the ASCII-table. Everything else is a `data` file.

In A1 we will extend our analysis to also handle such exotic file types as ISO-8859-1, UTF-8, as well as little-endian and big-endian UTF-16 text. In the

[1] For an explanation of the `(1)`-notation, see Appendix A.

end, certain files will still be regarded as `data`. For instance, a file that *only* contains some null-bytes will remain a mere `data` file.

We recommend using `printf(1)` with output redirection to generate files containing exotic characters (e.g., null-bytes). See also Appendix B for more information on how to generate exotic files directly from your shell.

1 API (15%)

Write a self-contained (one-file) C program, `file.c`, which when compiled to an executable `file` has the following API:

1.1. Let your `file` accept exactly one argument. If the given argument is a path to a file that exists, and the type of that file can be determined, write one line of text to `stdout`, and let `file` exit with the exit code 0 (`EXIT_SUCCESS`).

The line of text must consist of the given path, followed by a colon, space, and a guess of either `ASCII text`, `data`, or `empty`.

For instance, if there exists an `ASCII text` file called `ascii` in the same directory as your `file` binary:

```
$ ./file ascii
ascii: ASCII text
$ echo $?
0
```

The shell variable `$?` refers to the exit code of the previous command.

Similarly, if there exist corresponding files `data` and `empty`:

```
$ ./file data
data: data
$ ./file empty
empty: empty
```

Note: So far, the behaviour is exactly identical to `file(1)`. However, depending on your choice of characters, standard `file(1)` may report a *superstring*[2] of what your `file` reports. For instance, if you there are no line-break characters(`\n`), `file(1)` reports:

```
$ printf "Hello, World!" > ascii
$ file ascii
ascii: ASCII text, with no line terminators
$ ./file ascii
ascii: ASCII text
```

[2]Note that a *superstring* denote to a string that contains the related string but may also contain more. On the other side, a *substring* can contain a smaller part of the related string.

More formally, your `file` must report a non-empty *substring* of the type that `file(1)` reports. Note that the above example uses the `printf(1)` shell command and not the `printf` C function.

1.2. If instead `file` is called with *no arguments*, it prints the usage message "`Usage: file path`" to `stderr`, and exits with `EXIT_FAILURE`:

```
$ ./file
Usage: file path
$ echo $?
1
```

Note: The usage string printed for `file(1)` is far more complicated, not least because it supports far more file types than we can hope to cover. `file(1)` however, also writes a usage message (starting with the word "Usage:") to `stderr`, and exits with the exit code 1.

1.3. If we provide a path to a non-existing file, or some other I/O error occurs (e.g., someone rips out the USB-stick the file is on), behave as follows:

```
$ ./file vulapyk
vulapyk: cannot determine (No such file or directory)
$ ./file hemmelig_fil
hemmelig_fil: cannot determine (Permission denied)
```

In both cases, `file` must still print to `stdout` and exit with `EXIT_SUCCESS`.

To achieve the above behaviour, we provide the following function:

```
// Assumes: errnum is a valid error number
int print_error(char *path, int errnum) {
  return fprintf(stdout, "%s: cannot determine (%s)\n",
    path, strerror(errnum));
}
```

This utilizes the standard `strerror` function to print a standard string for a corresponding error number, as set by the standard I/O functions.

In particular, if an error occurs during `fopen(3)`, `fseek(3)`, `fread(3)`, or `fclose(3)`, the standard C library will set the so-called `errno` variable. This variable becomes available if you include the `errno.h` library. The `string.h` library then provides the function `strerror(3)`, which returns a string representation for an otherwise integer error code.

Note: `file(1)` behaves in a similar way: it exits with `0`, but writes rather different messages to `stdout`. `file(1)` does not regard invalid paths and I/O errors as errors on its part. This is a questionable design decision.

2 File Types (15%)

Your `file` should recognise the following file types:

`data`
: This file type must be reported in case no other type matches.

`empty`
: This file type must be reported if the file contains no bytes.

`ASCII text`
: This file type must be reported if all bytes belong to the following set:

$$\{\texttt{0x07}, \texttt{0x08}, \ldots \texttt{0x0D}\} \cup \{\texttt{0x1B}\} \cup \{\texttt{0x20}, \texttt{0x21}, \ldots, \texttt{0x7E}\}$$

3 Hints

3.1. Create some files to test `file(1)` and your `file` with first. (See Testing below.) Explore the behaviour of `file(1)`.

3.2. Discern between `empty` and `data` first. Handle the cases when your `file` is given too few (or too many?) command-line arguments. Only then proceed to also handle `ASCII text`. Write tests as you go along.

3.3. You might want to use an enum for all the supported file types and a constant array of user-friendly strings for each element of the enum. This way, your function for identifying the file type can use the enum, and the printing of the file type can happen at a later stage.

```
enum file_type {
  DATA,
  EMPTY,
  ASCII,
};

const char * const FILE_TYPE_STRINGS[] = {
  "data",
  "empty",
  "ASCII text",
};
```

3.4. Our reference `file.c` begins by including the following libraries, for the following reasons:

```
#include <stdio.h>  // fprintf, stdout, stderr.
#include <stdlib.h> // exit, EXIT_FAILURE, EXIT_SUCCESS.
#include <string.h> // strerror.
#include <errno.h>  // errno.
```

3.5. To create a non-readable file, you can use `chmod`:

```
$ printf "hemmelighed" > hemmelig_fil
$ chmod -r hemmelig_fil
```

The file is still writable, and you can also still delete it.

To make the file readable again:

```
$ chmod +r hemmelig_fil
```

4 Testing (30%)

The assignment has been carefully designed so that you can test your implementation by comparing its observable behaviour with that of the standard `file(1)` utility.

To this end, we hand out a shell-script `test.sh`. Run it as follows:

```
$ bash test.sh
```

This script will create a directory `test_files`, and fill it with sample files.

Your task: make it generate more interesting sample files.

The script will then compare the expected output (generated by `file(1)`, filtered to remove anything following `ASCII text`), and the `actual` output (generated by your `file` utility). If the files differ, the script will fail.

The comparison is done using the standard `diff(1)` utility. You should be familiar with diff's from Software Development. See also Appendix C.

4.1. Generate at least a dozen files for testing `ASCII text`.

4.2. Generate at least a dozen files for testing `data`.

4.3. Is the `empty` file type already tested?

5 Report (40%)

Alongside your solution, you should submit a short report; not exceeding 5 pages. The report should:

5.1. Describe how to compile your code and run your tests to reproduce your test results.

5.2. Discuss the non-trivial parts of your implementation and your design decisions, if any.

5.3. Disambiguate any ambiguities you might have found in the assignment.

Handout/Submission

Alongside this PDF file there is Zip archive containing a framework for the assignment. [3]

```
$ unzip src.zip
```

You might have to download the 'unzip' command-line utility. Another option is to temporarily rename this file to something ending in '.zip', use your standard unzipping utility, and then rename the file back to PDF. You can also download the ZIP archive uploaded alongside this PDF file[4].

You will now find a `src` directory containing the following:

`.gitignore`
: A suitable `.gitignore` file, should you choose to use Git.

`file.c`
: A skeleton file for your `file.c`.

`Makefile`
: The file that configures your `make` program.

  To compile the code with all the appropriate compiler flags, navigate your shell to the `src` directory, and simply execute `make`.

`test.sh`
: A shell script for testing your implementation.

Your task is to modify `file.c` and `test.sh`. You should hand in a ZIP archive containing a `src` directory, and the files `src/file.c`, `src/Makefile`, and `src/test.sh` (no ZIP bomb, no sample files, no auxiliary editor files, etc.).

To `make` this a breeze, we have configured the `Makefile` such that you can simply execute the following shell command to get a `../src.zip`:

```
$ make ../src.zip
```

Alongside a `src.zip` submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU-ids of your group members, one per line, and do so using *only* characters from the following set written using *standard alphabetical encoding*:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

Please, make sure that the file is UTF-8 encoded with UNIX line endings. Example can be found on Absalon.

[3] Alternatively, true to the spirit of this assignment, this file is both a PDF and a ZIP archive: it is a so-called polyglot file. If you are reading this, congratulations! You have successfully or opened a ZIP archive in your PDF viewer (or printed one). Now to unzip a PDF file: `unzip CompSys18-A0-polyglot.pdf`. PDF files are not ZIP archives in general; some formats are (e.g., DOCX); this is a hack.

[4] These are, in fact, the same file.

A `man(1)`

When we write `file(1)`, instead of just `file`, we mean that you can use the Unix-like command-line utility `man(1)` to look up the documentation of the command. For instance, try typing this at a Unix-like command line:

```
$ man 1 file
```

`man`-style documentation is organized into sections, and `file(1)` indicates that the documentation of `file` is in section 1. Often, `man` can guess the section you mean, and you can get away with things like this:

```
$ man file
```

`man`-style documentation is used not only for programs, but also other aspects of a Unix-like system. For instance, you will find a description of the file system hierarchy under `hier(7)`. Beware however, not all software developers keep their documentation up to date. You should.

Note: `man(1)` typically uses the `less(1)` to show the documentation, showing some basic controls at the bottom of the screen: To quit, type q. To get help, type h. Further, to scroll in the documentation using the ↑, ↓, PgUp, PgDn, Home, End.

B `printf(1)`

`printf(1)` prints a line of text to the standard output. You can use `printf` in conjunction with output redirection to create all sorts of files.

For instance, to create an ASCII file called `ascii`:

```
$ printf "Hello, World!\n" > ascii
$ file ascii
ascii: ASCII text
```

To create what will be regarded as a data file by `file(1)`, it suffices to insert an extra null-character somewhere in the file. Since the null-character is not printable (and hence, not writable), we will need an escape sequence (similar to standard C `printf(3)`):

```
$ printf "Hello,\x00World!\n" > data
$ file data
data: data
```

C `diff(1)`

`diff(1)` can compare files line by line. If you are familiar with the revision control system, git, you should be familiar with the format that `diff(1)` outputs with the command-line argument `-u` (for "unified context").

```
$ printf "Hello,\nWorld!" > x
$ printf "Hello,\nBrave New World!" > y
$ diff -u x y
--- x 1970-01-01 00:00:00.000000000 +0200
+++ y 1970-01-01 00:00:00.000000000 +0200
@@ -1 +1 @@
-Hello,\nWorld!
+Hello,\nBrave New World!
```

The exit code of `diff(1)` is non-zero if the files differ.

Note: `diff(1)` and `patch(1)` are rudimentary in a Unix-like development environment, although sometimes hidden behind high-level tools like `git(1)`.