Systems and Internet Infrastructure Security

Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

I/O (Part I)

Devin J. Pohly <djpohly@cse.psu.edu>

Input/output

- I/O is simply moving bytes into and out of a process's address space
- What's on the other side can vary:
  - terminal/keyboard (terminal I/O)
  - secondary storage (file and disk I/O)
  - devices in /dev
  - network (socket I/O)
  - another process (pipe I/O)

Buffered vs. unbuffered

- In many cases, I/O is automatically *buffered*
  - It may read more than requested and hold onto it, in the expectation that you will read again later (*read buffering*)
  - It may not write all bytes to the target immediately, so that it can send them in a larger chunk (*write buffering*)
- *Unbuffered* I/O will always read or write immediately

Blocking vs. nonblocking

- *Blocking I/O* – the call waits for the read/write to complete before returning
  - Default behavior
- *Non-blocking I/O* – the call always returns as soon as possible
- When fewer than the requested number of bytes are read/written, this is called a *short read* or *short write*
  - Blocking: usually indicates EOF (*end of file*) or an error
  - Non-blocking: happens all the time
  - Robust programs should always check to see exactly how much data was read/written

Terminal I/O

- There are three default terminal streams:
  - `stdin`: the main input to a program (e.g., a workload to simulate)
  - `stdout`: the main output of a program, often in a form that other programs could process
  - `stderr`: an additional output channel, used to communicate with the human operator (e.g., for error messages)
- Unless otherwise specified, these read from the keyboard and write to the terminal

Terminal I/O commands

- `echo`: prints literal text to stdout
  - `echo hello world`
- `printf`: prints formatted text to stdout
  - `printf '%x\n' 32`
- `cat`: copies file contents (or stdin) to stdout
  - `cat mtron.h`
- `tee`: copies stdin to stdout *and* to the given file(s)
  - `./prog | tee log.txt`

I/O redirection

- On the command line, "*redirection*" refers to using the contents of a file for input, output, or both
  - Output redirection sends the `stdout` of a program to a file:

```
$ echo "redirection demo" > demo.txt
$ cat demo.txt
redirection demo
```

  - Input redirection uses the contents of a file as `stdin`:

```
$ cat < demo.txt
redirection demo
$ tr r w < demo.txt
wediwection demo
```

  - You can also do both at the same time:

```
$ tr r w < demo.txt > elmerfudd.txt
$ cat elmerfudd.txt
wediwection demo
```

Pipes

- Redirection can only connect a file to a program
- To put two programs together, you use a *pipe*
  - Connects `stdout` of one program to `stdin` of the next
    - This is why `stdout` is often something another program can read (programs are filters)
    - `stderr` is still displayed on the screen for a human to read
  - Can form long chains (pipelines) of programs

```
$ printf '14\n21\n7\n4\n' > nums.txt
$ cat nums.txt
14
21
7
4
$ cat nums.txt | sort -n
4
7
14
21
$ cat nums.txt | sort -n | cat
4
7
14
21
$ cat nums.txt | sort -n | tac
21
14
7
4
```

Check this out...

- Even the shell is just a program that reads commands from `stdin`
  - Usually this means a human typing commands at a keyboard
  - But that's not required!
  - How does the example on the right work?
- Seriously... pretty much everything on the terminal uses `stdin`/`stdout`!

```
$ cat weird.sh
#! /bin/bash
echo echo hello world
echo 'a=3'
echo 'echo "a is $a"'
$ ./weird.sh
echo hello world
a=3
echo "a is $a"
$ ./weird.sh | bash
hello world
a is 3
```

File I/O

- File I/O provides access to a file within the filesystem
  - Located at a specific "path"
  - Keeps track of the current read/write offset in the file
    - Next read or write will begin at that position
  - All file I/O follows this pattern:
    1. Open the file
    2. Read/write the contents
    3. Close the file

Locating files for I/O

- Two kinds of path: absolute and relative
- An *absolute path* specifies the directories and filename starting from the filesystem root "/":
  - `/home/djpohly/docs/cmpsc311-s14/www/Makefile`
  - Absolute paths *always* start with a slash
- A *relative path* gives the directories and filename starting from (relative to) the current directory:
  - Suppose current directory is `cmpsc311-s14`
  - `www/Makefile` is the same file as above
  - You can also use `..` to go up to the parent directory
    - `../../bin/args` would be `/home/djpohly/bin/args`

High-level I/O functions

- One way to do I/O is to use the high-level functions from the C standard (found in `<stdio.h>`)
  - These use the `FILE` structure: a set of data items that are created to manage input and output for the programmer
    - High-level abstraction that hides some of the details of file I/O
    - You will only deal with `FILE *`, often referred to as a *stream*
  - Frequently used for reading and writing ASCII text

```
(gdb) p *file
$1 = {_flags = -72539008, _IO_read_ptr = 0x0, _IO_read_end = 0x0,
_IO_read_base = 0x0, _IO_write_base = 0x0, _IO_write_ptr = 0x0,
  _IO_write_end = 0x0, _IO_buf_base = 0x0, _IO_buf_end = 0x0,
_IO_save_base = 0x0, _IO_backup_base = 0x0, _IO_save_end = 0x0,
  _markers = 0x0, _chain = 0x7ffff7dd41a0 <_IO_2_1_stderr_>, _fileno =
7, _flags2 = 0, _old_offset = 0, _cur_column = 0,
  _vtable_offset = 0 '\000', _shortbuf = "", _lock = 0x6020f0, _offset
= -1, __pab1 = 0x0, __pad2 = 0x602100, __pad3 = 0x0, __pad4 = 0x0,
  __pad5 = 0, _mode = 0, _unused2 = '\000' <repeats 19 times>}
```

Opening a stream

- The fopen function opens a file for I/O and returns a pointer to a `FILE` structure:

```
FILE *fopen(const char *path, const char *mode);
```

  - `path`: string containing the absolute or relative path to the file to be opened
  - `mode`: string describing the ways the file will be used
  - For example:

```
FILE *file = fopen(fname, "r+");
```

  - Returns a valid `FILE *` if successful, otherwise NULL
    - You don't have to allocate or deallocate the `FILE` structure; this is done automatically by the implementations of `fopen` and `fclose`

fopen modes

- `r`: Open text file for reading. The stream is positioned at the beginning of the file.
- `r+`: Open for reading and writing. The stream is positioned at the beginning of the file.
- `w`: Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- `w+`: Open for reading and writing. The file is created if it does not exist, otherwise it is truncated.
- `a`: Open for appending (writing at the end of file). The file is created if it does not exist.
- `a+`: Open for reading and appending. The file is created if it does not exist.

Reading stream data

- There are two dominant ways to read text from a stream: `fscanf` and `fgets`
  - `fscanf` reads data from a file exactly like `scanf`:

```
int x, y, z;
if (fscanf(file, "%d %d %d", &x, &y, &z) != 3)
    return 1;
printf("Got coordinates [%d,%d,%d]\n", x, y, z);
```

  - `fgets` reads a line of text from a file (including the newline), up to a certain number of bytes:

```
char str[128];
if (fgets(str, 128, file) == NULL)
    return 1;
printf("Got line: %s", str);
```

Writing stream data

- There are two dominant ways to write text to a stream: `fprintf` and `fputs`
  - `fprintf` writes data to a file exactly like `printf`:

```
int x = 3, y = 3, z = -5;
fprintf(file, "%d %d %d\n", x, y, z);
```

  - `fputs` writes a string to the file:

```
char str = "fputs demo\n";
fputs(str, file);
```

Flushing a stream

- Standard stream I/O is usually buffered
  - Notable exception: `stderr` is unbuffered by default
- `fflush` empties the buffer:

```
int fflush(FILE *stream);
```

  - For an output stream, this will make sure everything is written from the buffer to the stream before returning
  - For an input stream, this function ***does not do what you think it does***.
- `fflush(NULL)` will flush all open output streams

Closing a stream

- The `fclose` function closes the file and releases the memory associated with the underlying `FILE` structure
  - Remember: since this frees memory to which you have a pointer, it's a good practice to set that pointer to `NULL`

```
fclose(file);
file = NULL;
```

  - Note: `fclose` implicitly flushes any buffered data to the file. (This is what you'd expect it to do.)

Terminal I/O streams

- Remember our three default terminal I/O *streams*?
- The type of `stdin`, `stdout`, and `stderr` in a C program is `FILE *`
  - Note: you don't need to open or close these to use them
- You can use the stream functions for terminal I/O as well
  - `scanf(...)` is the same as `fscanf(stdin, ...)`
  - `printf(...)` is the same as `fprintf(stdout, ...)`
  - To print messages to the error stream, use `fprintf(stderr, ...)`
  - Using `stderr` for debug messages is better than the default `stdout` that `printf` uses (why?)

Putting it all together

```c
#include <stdio.h>

int read_demo(FILE *file) {
    // Read until we reach the end
    while (!feof(file)) {
        // How many lines?
        int lines, i;
        if (fscanf(file, "%d\n", &lines) != 1)
            return 1;
        printf("%d lines:\n", lines);

        // Print them out
        char str[128];
        for (i = 0; i < lines; i++) {
            if (fgets(str, 128, file) == NULL)
                return 1;
            printf("%2d: %s", i + 1, str);
        }
    }

    return 0;
}

// continued on next slide...
```

```
$ cat /tmp/fopen.dat
5
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a yellow wood, and I--
I took the one less traveled by,
And that has made all the difference.
3
Here I stand, in the light of day.
Let the storm rage on...
The cold never bothered me anyway.
```

Putting it all together

```c
// ... continued from previous slide

void write_demo(FILE *file) {
    printf("Adding Shakespeare...\n");
    fprintf(file, "%d\n", 2);
    fputs("To be or not to be:\n", file);
    fputs("That is the question.\n", file);
}

int main(int argc, char **argv) {
    FILE *f = fopen("/tmp/fopen.dat", "r+");
    if (f == NULL) {
        perror("fopen");
        return 1;
    }

    // Opened for reading and writing
    read_demo(f);
    // read_demo leaves us at EOF,
    // so this will append
    write_demo(f);

    fclose(f);
    return 0;
}
```

```
$ ./fdemo
5 lines:
 1: I shall be telling this with a sigh
 2: Somewhere ages and ages hence:
 3: Two roads diverged in a yellow wood, and I--
 4: I took the one less traveled by,
 5: And that has made all the difference.
3 lines:
 1: Here I stand, in the light of day.
 2: Let the storm rage on...
 3: The cold never bothered me anyway.
Adding Shakespeare...
$ tail -4 /tmp/fopen.dat
The cold never bothered me anyway.
2
To be or not to be:
That is the question.
```