THE UNIVERSITY OF SYDNEY

COMP2129 Week 7 Tutorial

Pipes, signals, and floating-point

File descriptors

Processes running inside UNIX have a table set aside for keeping track of what files they have open, called the *file descriptor table*. The entries in the table usually start with 0, 1, 2 (standard in, out, and error), and then any subsequently opened files are stored at at higher indexes. When a process opens a file (or other file-like object), the first free file descriptor is used. For example, if the numbers 0, 1, 2, 4, 6 were in use, the next open file would take the value 3.

The **`read()`** and **`write()`** system calls can operate on bare file descriptors, rather than the **`FILE *`** streams provided by the C standard library. Check out their manpages in section 2: **`man 2 read`**, and **`man 2 write`**. For example, if you include the following lines in a program, you should see the same output as a similar call to **`printf()`**, since file descriptor 1 is standard out:

```
1  char msg[] = "Hello, world!\n";
2  write(1, msg, strlen(msg));
```

Similarly, the **`read()`** command can read bytes from standard in (file descriptor 0) into a buffer:

```
1  char buf[1024];
2  int nread = read(0, buf, sizeof(buf));
3  printf("%d bytes read\n", nread);
```

`read()` returns the number of bytes read, and **`write()`** returns the number of bytes written. These functions are too low-level for most purposes, and have many tricky nuances to deal with. Later in the tutorial, the **`fdopen()`** command is used to convert a file descriptor into a **`FILE *`**, something a lot nicer to deal with.

Other functions which operate on file descriptors, such as **`close()`**, **`dup()`**, and **`dup2()`**, will be explored in the rest of this tutorial.

Exercise 1: Parent-child process communication

The standard way of communicating between parent and child processes is through something called a *pipe*. A pipe is a data flow through the operating system kernel: one end is writable, and anything written there will show up on the other end of the pipe. Pipes (as well as all other file descriptors) are preserved across actions like **`fork`** and **`exec`**, allowing a parent and child process to share a pipe.

A call to create a pipe looks like this:

```
int pipefd[2];
if (pipe(pipefd) < 0) {
  perror("Could not create pipe");
  // Handle error
}
```

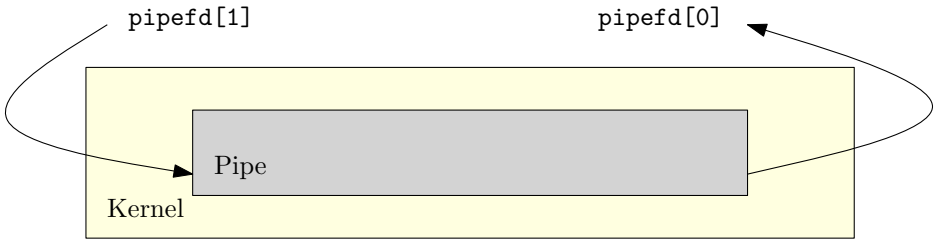

Figure 1: Buffered contents of the pipe are stored by the operating system kernel. Data written to **`pipefd[1]`** appears on **`pipefd[0]`**.

The integers **`pipefd[0]`** and **`pipefd[1]`** are the file descriptors for the pipe. Calls to **`read()`** and **`write()`** may *block* until input/output is possible. (Blocking means a function which may not return for a while, i.e. blocks the code at that line). For example, trying to read from a pipe will block until there is data to read. Writes to a pipe may block when the kernel's buffer allocated for the pipe is full.

Write a program which does the following:

1. Creates a pipe, and forks itself into a parent and child process.
2. In the parent process, closes the read end of the pipe using **`close()`**, and then writes a message to the child over the pipe using the **`write()`** system call. Include a call to **`printf`** before writing to the pipe, so that you can watch what's happening. The parent should **`close()`** the pipe when it's finished with it.
3. In the child process, closes the write end of the pipe using **`close()`**, and then waits for a message from the parent by trying to **`read()`** from the pipe. After recieving this message, the child should use **`printf`** to write a message out before exiting.
4. Try playing around with the timing of things, to convince yourself that the call to **`read()`** really does block until the parent writes something. For example, include a call to **`sleep(1)`** before the parent writes to the pipe.

An example program might have the following output:

```
Parent: Sending "Luke, I am your father." to the child.
Child: Nooooo!!!
```

Exercise 2: Reading the output of a process

There's nothing special about file descriptors 0, 1, or 2, besides the fact that these are where **`stdin`**, **`stdout`**, and **`stderr`** go. The **`close()`** system call can remove an entry from the file descriptor table, freeing up a number for reuse. The **`dup()`** (short for *dup*licate) system call makes a copy of a file descriptor into the lowest available index in the table. Using these together allows the programmer to replace what "standard out" is:

fd	Destination
0	Terminal
1	Terminal
2	Terminal
3	(empty)
4	(empty)

Program start

fd	Destination
0	Terminal
1	Terminal
2	Terminal
3	**`pipefd[0]`**
4	**`pipefd[1]`**

`pipe(pipefd)`

fd	Destination
0	Terminal
1	(empty)
2	Terminal
3	**`pipefd[0]`**
4	**`pipefd[1]`**

`close(1)`

fd	Destination
0	Terminal
1	**`pipefd[1]`**
2	Terminal
3	**`pipefd[0]`**
4	**`pipefd[1]`**

`dup(pipefd[1])`

Table 1: The file descriptor table over the course of the program.

After the process table has been rearranged like this, any writes to standard out will actually go through the pipe instead. Since the file descriptor table is preserved across the system calls **`fork`** and **`exec`**, any program which is now executed will be writing to the pipe, instead of the terminal, for its standard output.

Write a program which reads back the contents of the **`ls -l`** command through a pipe, by following these steps:

1. Create a pipe, and fork off a child process.
2. In the child process, close the read end of the pipe, and replace file descriptor 1 with the write end of the pipe. Then use the **`execlp`** function to replace the child with **`ls -l`**.
3. In the parent process, close the write end of the pipe, and convert the pipe to a file stream by using **`fdopen()`** (see the note below). Then read the contents of the pipe line by line using **`fgets`**, giving each line a custom prefix (so you know the program is working).

The output of the program might look something like this:

```
Line 1: -rwxr-xr-x  1 jag  staff   8988 10 Apr 09:43 a.out
Line 2: -rw-r--r--  1 jag  staff  25005 10 Apr 12:29 pipe.pdf
Line 3: -rw-r--r--  1 jag  staff    680 10 Apr 12:31 test.c
```

To convert a file descriptor to a file stream (**`FILE *`**), use the **`fdopen()`** function:

```
FILE *fp = fdopen(pipefd[0], "r");
```

Now the stream **`fp`** can be used just like any other file-like object we've used up to this point in the course. In particular, **`fscanf`**, **`fgets`** and so on will work. Converting to a file stream hides many of the "ugly" parts of dealing with file descriptors: we rely on the C standard library to do the heavy lifting for us.

Exercise 3: Handling Signals

Programs running in the UNIX environment must handle *signals*, a form of interprocess communication. Some common signals are **SIGINT**, sent to programs when **Ctrl+c** is entered into a terminal, **SIGTERM**, send to stop processes (for example the shell comman **kill <pid>** sends **SIGTERM** by default), and also the infamous **SIGSEGV**, also known as "segmentation fault". Some of these signals can be caught and handled by programs: you may have noticed that while most shell commands can be killed with **Ctrl+c**, editors like **nano** and **vim** can't. This is because they have handlers in place for **SIGINT**, instead of the default handler (killing the process).

In this exercise we will implement a signal handler for **SIGTERM**, to print out a "goodbye" message before exiting. A signal handler is a function taking an integer (the signal to be handled) and returning nothing.

```
#include <signal.h>
void handle_sigterm(int sig);
```

Sometime during your program's execution, you should register the signal handler using the **sigaction** function. Type **man 3 sigaction** to get a summary of the function, along with a list of possible signals that can be handled. Here we set up a basic struct containing information to be passed to **sigaction**:

```
struct sigaction sa;
sa.sa_handler = handle_sigterm;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);

if (sigaction(SIGTERM, &sa, NULL) == -1) {
  perror("Could not register handler");
  exit(1);
}
```

After the signal handler has been registered, when **SIGTERM** is delivered to the process, the process will jump to the signal handler, and when the signal handler returns, the process will jump back to where it was before the signal was delivered.

1. Create a program called **mysig.c** which registers a handler for **SIGTERM** and then goes to sleep for a long time, using the **sleep()** function (see below). Make sure that the signal handler for **SIGTERM** prints a message and then exits the program.

2. Run the program. In another terminal, run **ps aux**, and pipe the output through **grep mysig** to find the process ID. Send **SIGTERM** to the process by using the **kill** shell command.

3. Modify your program to register a signal handler for **SIGINT** in addition to **SIGTERM**. Make the handler for **SIGINT** prints out a message but does not exit the program. Try killing your new program using **Ctrl+c**: it should not work. Send **SIGTERM** to it to kill it.

4. There are some signals which may not be caught: one such signal is **SIGKILL**. Try to catch this signal, and observe what error message **perror** reports.

Hint: You can put an infinite pause in your program by using `for (;;) sleep(1);`

Floating-point oddities

If you want to represent "real numbers" (in the maths sense) on a computer, you'll probably first reach for floating-point numbers, represented as the C types `float` or `double`. These numbers are essentially a fixed width (32-bits for `float`, 64 bits for `double`) set of bits, reinterpreted as numbers in scientific notation. For this tutorial, we'll restrict our focus to 32-bit floating point numbers, however the same ideas hold for any size.

It's most accurate to think of floating-point numbers as being in scientific notation. Computers use base two, but thinking of computations happening in base 10 is actually fine, and will lead to the same conclusions about where and why odd behaviour occurs. A floating-point number is made of 32 bits. One bit is reserved for the *sign* s (positive or negative) of the number. 8 bits make up the *exponent* e, and the rest of the 23 bits are called the *mantissa* m. The value represented by the triple (s, e, m) is

$$(-1)^s \times 1.m_1 m_2 \ldots m_{23} \times 2^{e-127}$$

Since numbers are represented in scientific notation, there are lots of "bad things" which can happen, mostly due to loss of precision. The following exercises will explore a few of these. For these exercises, you should always be using the type `float`, and not mixing types. If you need to add an integer `i` to a `float`, first save the `int` as a `float` before adding it.

You may like to use the `%g` format string for printing out floats using `printf`.

Exercise 4: Machine Epsilon

How is the number 1 represented in scientific notation? Looking at the equation above, we can see the simplest way is to set $s = m = 0$, and $e = 127$. So, what happens if we try to add a very small number to 1? How small does the number have to be such that it is "totally insignificant" compared to 1, and will not be able to fit in the mantissa of 1? (Can you make a guess at what it is?)

Find, to within a factor of 2, the largest `float` (let's call it the *machine epsilon*) which can be added to the number 1, and not change its value. You may like to investigate the log-base-two of the machine epsilon, and confirm that it agrees with your theory. What does this tell you about testing the equality of floating-point numbers. Is it a good idea?

Exercise 5: Addition is not associative

In mathematics, we say an operation like $+$ is *associative* if $(a + b) + c = a + (b + c)$ for all numbers a, b, c, i.e. the bracketing, or order of application does not matter. Unfortunately, this is not the case for floating-point numbers in general. Because adding a tiny number to a large number may not have an effect on the large number, but adding up many tiny numbers can make a large number, addition is not associative.

Write a program which adds up the first one million numbers in ascending order into one floating-point variable. Add up the first million numbers in descending order into another variable. Print out and compare these two sums.

Exercise 6: What's in a float?

The following union can help you take apart a float (on little-endian machines):

```
#include <inttypes.h>

union Floatbits {
  float f;
  struct {
      uint32_t mantissa : 23;
      uint32_t exponent : 8;
      uint32_t sign : 1;
  } parts;
};
```

Make a table of values for some interesting float values, such as $0, 1, 2, -1, 1.5, 1.75, 2 \times 10^{30}$, and so on. Your table of values should contain the following columns:

- Float value (the numerical value of the float).
- The sign bit.
- The mantissa (possibly in binary representation).
- The exponent *value* (the value of the exponent field, minus 127).

Make sure the output of your program agrees with your interpretation of how floating-point works. There are also a few special values which can be represented by floats, such as $\pm\infty$, and **`NaN`** (Not a Number). Look up how to these are defined, and how to get a hold of them in your code (you may have to directly fill out the parts of the float yourself).

Exercise 7: Extension: Pipe like a shell

In a previous exercise, input was read from an **exec**'d program back to the parent program, through a pipe. However, there is nothing stopping us from simply dupping the read end of the pipe into the standard in of a second child process, and calling **exec** again: this is the nature of a shell pipe. To implement a shell, the programmer does this for arbitrarily many programs: here we will consider just two.

We will attempt to re-create the following line of bash, only using C:

```
$ sort <filename> | uniq
```

The programs we will be executing are **sort** (with one argument: the file we want it to open), and **uniq** (with no arguments: it gets all its input from the output of **sort**).

Write your program **sortuniq**, which takes a single argument as input: the filename to provide **sort**. The sequence of calls is shown in the diagram below.

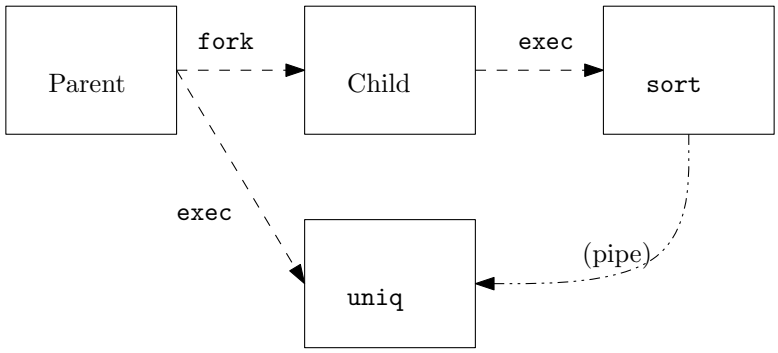

Figure 2: The parent forks one copy of itself. The child execs into **sort**, and the parent execs into **uniq**

Consider the following extensions to this task:

- How would you redirect a file into **sort**, rather than giving **sort** a command line argument? Keep in mind that *any* file descriptor can be switched in for standard in. You may like to look at **man 2 open**.
- Can you redirect output in a similar way to redirecting input?
- Suppose you are writing a shell. What data structures would you use to represent a pipeline, so that more than two programs could be connected by a pipe? Can you see a (perhaps recursive) way to generalise this exercise to an arbitrary number of programs? (Aside: usually when running a long pipeline in a shell, like `prog1 | prog2 | prog3 | prog4`, only `prog4` is a direct child of the shell, while the first three are direct children of the last. The shell then **wait**s on the last program.)

Hint: The **dup2()** system call takes two arguments, and works like a **close** followed by a **dup**.