Lecture 11: Redirection

What is Redirection?

We've covered this a bit already, but the simplest example of redirection is:

```
ls > temp_file.txt
```

So, what actually happens when you use the '>' or '<' operators? Well, to understand this we need to first have a think about streams.

Streams

As the name suggests a stream is a stream of something. In computer science (specifically Unix and C), it is usually a stream of characters. Simply put a stream of characters is an arbitrary number of characters in a sequence of some kind. Most files can be easily conceived of as a stream. One program might produce these character (i.e. an output stream) and one program might absorb these characters (i.e. an input stream).

In Unix, there are some **default streams** which are the ones you should be most familiar with. These are Standard In, Standard Out and Standard Error (this last one might be new). Each of these default streams comes with a **file descriptor** which is basically like a number which streams are assigned to (because computers like numbers rather than words). The file descriptors of the basic streams are below.

File Descriptor	Stream Name	Connection
0	stdin	Keyboard
1	stdout	Screen
2	stderr	Screen (unbuffered)

File Descriptors

Ok, perhaps my previous definition was a bit minimal. More precisely, a **file descriptor** is a connection form a **process** to a **file**. Each file descriptor is identified by a unique number (0, 1, 2 etc). Each process has a list of its **file descriptors**. A simple way of thinking about this is that if you have two different programs running, each program may read from and write to different locations. One program might have a file open, while another does not. Thus each program needs to know what **files** (remember this means practically anything) it has a relationship with and what that relationship is. This is all managed via **file descriptors**.

Now, individual processes do not have direct access to files. Instead the kernel stores a **filetable** which then refers to an INode table. The **file descriptor** of a process maps to this **filetable** which in turn maps to the **INode table** where the file actually exists. The **filetable** manages the modes of the files too (such as read/write). A simple example is that **standard in** is a **read-only** 'file'.

For reference, when creating new file descriptors, Unix starts from zero and works its way up.

What is Redirection? #2

So, knowing now what a **stream** is and what a **file** is and what a **file-descriptor** is, describing redirection is a bit easier.

A **process** has access to some file descriptors. These refer to some streams. The default ones are standard in and standard out. However, they do not need to be. What if, instead of reading from **standard in**, you simply read from a file somewhere. This is the essence of redirection. However, to make sense of this in a technical sense, we know that **standard in** has a file descriptor. What if, we replaced that file descriptor (by default 0) to instead point at a file. Then whenever we make a call to **standard in** (such as fgetc) we are instead reading from the file as if it were a stream of characters.

So how does this transfer take place?

Well, it is possible to 'close' streams. This is a bit like hanging up the phone to free up the line. By default, each process has two 'phone lines' open (standard in and standard out). You could hang up one of these 'phones' and then call a file instead.

A C Example of Redirection.

Let us start with imports:

```
#include <stdio.h>  // For writing
#include <fcntrl.h> // For <we'll see>
```

Now the main code.

```
int main()
{
      int fd;          // This will be important later
      char line[100];  // Somewhere to store things
      fgets(line, 100, stdin);
      printf("%s", line); // Read and print
      fgets(line, 100, stdin);
      printf("%s", line);
      fgets(line, 100, stdin);
      printf("%s", line);

      // stdin just disappeared
      close (0);
      fd = open("<a path>", 0_RDONLY); // <= This is a flag
      // Die gracefully if file doesn't exist
      if (fd != 0)
      {
            fprintf(stderr, "Could not open data as fd 0\n");
            exit (1);
      }
      fgets(line, 100, stdin);
      printf("%s", line); // Read and print x3
      fgets(line, 100, stdin);
      printf("%s", line);
      fgets(line, 100, stdin);
      printf("%s", line);
      return 0;
}
```

This code can be broken into four main parts. Part 1: Read some stuff from standard in and then print it out. Part 2: Close standard in (0 is the file descriptor for standard in). Part 3: Open a file in read-only

mode as a stand in for standard in (also check if it worked). Part 4: Read from the file (rather than standard in) and print the result to the screen.

What is interesting here is that we are still explicitly reading from **standard in** or at least, the file descriptor assumed to be **standard in**. The trick is that now when the program asks for input from **standard in** it is instead going to receive information from the specified file. There are ways of explicitly reading from files using a different file descriptor (i.e. fd = 3 or higher) but in this case that is not what we are doing. **fgets** reads from file descriptor 0 by default (which is usually standard in). In this case, fd=0 is simply not **standard in** any more.

One problem with this code is that it contains an assumption. When we close fd=0 and then open a file, we are assuming that the operating system happily gives that file the **file descriptor** zero (because it should be free now). However, this is a dangerous assumption. It would be nicer if we could explicitly ensure that we get the correct **file descriptor**.

dup2

dup2 is a function specifically for the purpose of ensure correct copying of **file descriptors**. It stands for 'duplicate' which describes its purpose. It duplicates **file descriptors**. The signature is as follows:

```
dup2(old_fd, new_fd);
```

What **dup2** does is that it makes 'new_fd' a copy of 'old_fd'. Now 'new_fd' will point where 'old_fd' did.

```
...
close (0); // stdin just disappeared
fd = open("<a path>", O_RDONLY); // <= This is a flag
error_thing = dup2(fd, 0); // Closes 0, fd is now 0
if (error_thing != 0)       // Die gracefully
{
      fprintf(stderr, "Could not duplicated fd to 0\n");
      exit (1);
}
close(fd);       // Closing the file (from fd's perspective)
...
```

If we replace the close/open code in the first example with this code, we can ensure a smooth file descriptor transition. Here we create a new file descriptor… let's say it is '3'. Then we close stdin, replacing it with 'fd'. Now, our further code should still work. Finally we close fd (i.e. the '3'). Everything should now work with nothing left hanging.

For reference, there is also a program: **dup** (no 2). This program works like **dup2** but without doing the close of **new_fd**. In general, using **dup2** makes more sense than **dup**.

File Descriptors and the Exec group of Functions

One neat trick is to realise that **file descriptors** can last through calls to **exec**. Let us look at an example:

```
int main()
{
      int id;
      id = fork();
      if (id == 0);
      {
            // Child
            {
                  close(1);
                  fd = creat ("ls_output", 0644);
                  execlp("ls", "ls", NULL);
                  perror("execlp");
                  exit(1);
            }

      }
      else if (id != 0)
      {
            // Parent
            wait (NULL)
      }
}
```

Here we do a normal fork, splitting into two different paths. In the parent, we simply wait. In the child, we close **stdout** and then open a file. That file will by default take the **file descriptor** '1' (which is **stdout**). Now when **execlp** runs 'ls' the output of 'ls' will go to the file we have created.