## Practical 3 Basic I/O Programming in Unix

The first two practicals were intended to help you become familiar with the Unix operating system environment. This included a presentation of basic commands, filespace navigation, and using editor programs.
You should be able to answer these ten quick questions:-

1. How do you get a list of all processes running on your computer?
2. What symbols are used to represent:- 1) the root directory; 2) your home directory; and 3) the current directory?
3. What is $PATH ?
4. How do you get the full pathname of your current directory?
5. What is a command shell program?
6. How do you run a command in parallel with the shell process so that the shell is not blocked from accepting further commands while that command is running, for example if you wanted to run gedit and still be able to enter shell commands while gedit was open?
7. How do you tell the shell to redirect the output of a process to a file instead of displaying it on the screen?
8. How do you tell the shell that it should connect the output of one process to become the input of another process?
9. What is a shell script?
10. How do you pass arguments to a script?

### Using Operating System Functions

You have already implicitly created processes, done I/O and interprocess communication in Unix by simply typing commands to the shell in earlier practicals. The shell's job is to identify and load the binary images listed in your command from the file system by using the $PATH environment variable, and to automatically create a new process in the system to execute each program listed in your command. The shell also automatically sets the input and output devices for each process that it creates either by default or according to other specifications in your command.

So if you type "`du -a ~`" to see the disk block usage of files in your home directory tree, the shell creates a new process and the binary image of "`du`" is loaded into it and the process inherits the input and output devices of the shell by default (the keyboard and xterminal text window) as no other instructions are included. So the output is displayed in the shell's window while the shell waits for the "`du`" process to end. When it is finished, the shell puts out the $ prompt to indicate that it is ready to accept more commands.

If you submit the command "`du -a ~ | grep myfile`", the shell will create two concurrent processes and it will change the standard output stream of the "`du`" process to be connected to the standard input stream of the "`grep`" process through a pipe mechanism. The "`du`" process therefore does not output anything to the xterminal window but all its output is instead sent to the "`grep`" process. The "`grep`" process reads this input searching for the string "`myfile`" and outputs any lines, which contain that string, to the shell's xterminal window. **So the shell's command syntax hides all that complexity of doing process creation and handling pipe stream communication between processes and I/O redirection.**

One of the tasks of an operating system is to make the system easier to use and the shell fulfills that objective to some extent in this way.

**Purpose of Forthcoming Practicals**

In the next set of labs, we want to show how we can explicitly make these kinds of system calls in programs, giving you more of an insight into how the shell program itself works and how to write more flexible programs that use operating system calls to use I/O devices or create other processes to communicate and exchange data in different ways.

The C language will be used in some of these practicals to demonstrate some Unix features. Although you may have no experience with C, it is very similar in basic syntax to Java (Java syntax was derived from it) and should present no problems to you in terms of understanding and reading the code. C is a procedural language rather than an Object Oriented Language which means that a program is described by a collection of functions/routines derived from a system of stepwise refinement of a sequential algorithm. C functions are like methods within Java objects except they are not associated with any object structure encapsulating them and only have one instance within the process. We will present you with explanations of the code you will see and use. The practical also serves the utility of giving you some experience with writing and compiling C code so try and look on that as secondary benefit.

The reason we use C and not Java is because the Java language programs are executed on an implementation of the Java Virtual Machine and not directly on the host operating system and architecture. C programs are executed directly on the host operating system and C executable binary files are compiled for and executed directly by the host architecture. This makes it easier to explain aspects of Unix using example C programs rather than Java. Doing I/O is a bit awkward in Java.

**Getting Started with C**

Make cs240 your current directory. Create a new directory here called practical3 for files relating to today's practical. Change your current directory to practical 3.
Open an editor and create the C program given below and save it as "p3a.c".

```
#include <stdio.h>
int main()
{
     printf("Hello World\n\n");
}
```

The program must now be translated from C to machine code before it can be loaded as a process and executed. We use a C compiler program to do this and our compiled code must be linked with any already compiled code from libraries used in our program all joined together to create a single executable output file. The command below executes the cc compiler to compile our program file p3a.c and create an output file called p3a.

To Compile the program and create the executable file enter

```
cc p3a.c -o p3a
```

and run the executable file by entering

```
./p3a
```

The program uses a standard library I/O function (`printf`) to write out a simple message to the default output device of the process. To use any of the standard C language I/O defines and functions you must include the stdio library's header file in your C programs with the following statement included at the top of the program.

```
#include <stdio.h>
```

When the compiler comes across the identifier "printf" it will now know what that identifier means because its definition is included from the file stdio.h.
The string to be printed by printf contains the escape sequence "\n". This means to print a carriage return (new line) to the output. printf is a formatted output function. The string parameter to be printed contains various formating codes for different types of data items that are to be printed to screen.

You can compile code on line, in many languages including C at the following web site http://compileonline.com/ You might find that a useful learning environment. Try compiling our C program p3a.c on that site. Replace the default code with p3a.c

---

Open an editor and enter the following code saving it as p3b.c
Compile the program and execute it.

```
#include <stdio.h>
int main()
{
    char name[30];
    int age;
    int n;

    printf("What's your name and age?\n");
    scanf("%s %d", name, &age);
    printf("Hello %s, You are %d years old\n", name, age);
}
```

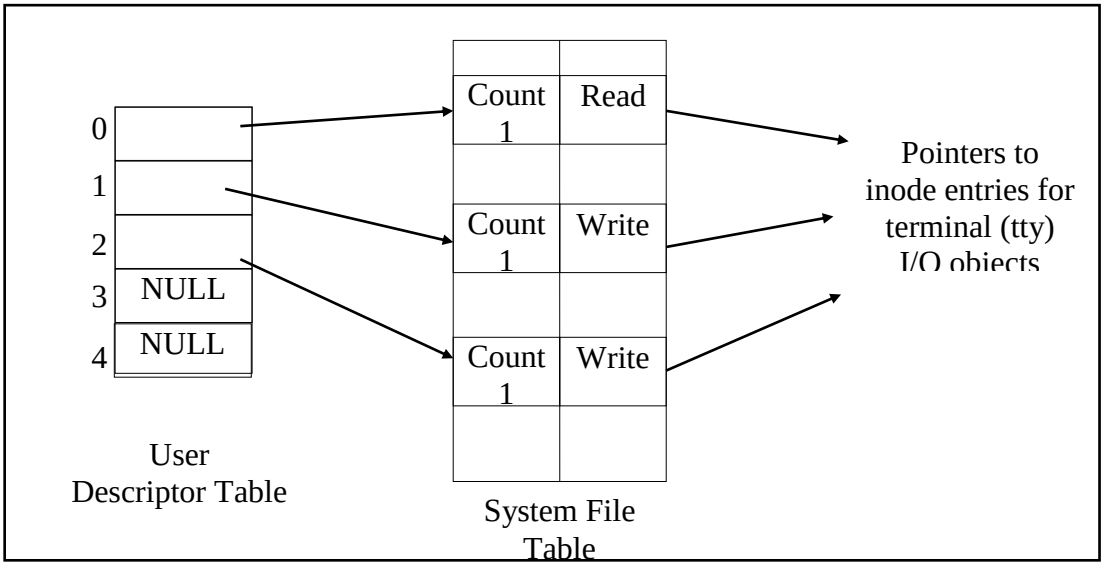
**Explanation**

The program defines an array of characters called `name` and an integer variable called `age`.
`scanf` is a formatted read function, it is being asked to read a sequence of characters and turn it into a string followed by an integer and to place that data in variables `name` and `age` respectively. We then see a further use of the formatted print statement `printf` to print a string of mixed data types to the output.

**Basic I/O Programming in Unix**

Before we go on to look at interprocess communication and other operating system features of Unix, we will first cover a little bit on how a process carries out basic input/output in Unix.

One of the great strengths of Unix is the consistency with which all input and output can be performed with a wide variety of I/O objects. I/O Objects, such as files and communication channels are manipulated in Unix by generic system calls such as **read** and **write**. Each I/O object in use by a process is assigned a number which identifies an associated system structure known as an *I/O descriptor*. (See Diagram below). The number of the I/O object is used by the process as a parameter to system calls, such as read and write, to enable them identify the I/O object which is to be manipulated. The I/O descriptor stores information relating to the current state (e.g. reading position) of its I/O object. All I/O descriptors are stored in a system structure called the *system file table*.




Each process has its own unique table of pointers, which identify descriptors in the central system file table, relating to I/O objects which that process has opened for access. This table is known as a User Descriptor Table above. The index (*0..n-1*) of each pointer in the user descriptor table is the number with which the process identifies each of its *n* I/O objects.

By default, processes have a standard input stream (e.g. keyboard) and a standard output stream (e.g. terminal text window). The first two pointers to I/O objects in a user descriptor table are normally used for this purpose. A third pointer is generally used as well to indicate the place where system error messages will be written (e.g. terminal text window also). Entry 0 (**`stdin`**) is the default input stream, Entry 1 (**`stdout`**) is the default output stream and Entry 2 (**`stderr`**) is the default error output stream.

The standard C functions of the *stdio* library, **`printf`** and **`scanf`** use devices **`stdout`** and **`stdin`**, respectively, by default. **`printf`** is a formatted write function similar to the **`println`** method of a `Java` **`PrintStream`** `object`. **`scanf`** is an input function.

**Read and Write**

The **read** and **write** system calls are a lower level generic means of performing I/O than printf and scanf. They are lower level because they require the programmer to supply extra parameters and parse the characters read and assemble them into meaningful data. The formatted I/O functions do this automatically making life a bit easier. On the other hand, reading and writing at the character or byte level has the benefit of extra flexibility and parsing things the way you want to which makes the calls more generic for other I/O purposes.

The syntax of the write system call is

```
write(fd, buffer, count);
```

where fd is an integer index into the file descriptor table (see earlier diagram), buffer is the address of a character string and count is the number of bytes of that string to be written to the device fd.

Compile and run the following C program, as p3c.c
It gives the same output as our first program p3a.c, but uses the write system call instead of printf:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
     char buffer[14]="Hello World\n\n";

     write(1, buffer, 13);
}
```

Modify the program so that the message is also displayed on the standard error output stream. Change the file descriptor parameter to that of **stderr**.

Note the difference between the `printf` statement and the lower level `write` statement. The `write` statement requires the programmer to identify the output device and to specify exactly how many characters to write. The `printf` statement always writes to device 1 and writes out the characters of a string from a starting address (the variable's name) until the NULL character is reached.

The syntax of the read system call is as follows:-

```
nbytes = read(fd, buffer, count);
```

where fd is an integer index into the file descriptor table, buffer is the address of a string buffer to be filled and count is the number of bytes to read.

(NB. Make sure the buffer is big enough to store count bytes.)

The read function returns the actual number of bytes read from device 0 (stdin), which may be less than count if end of input is reached. This value is stored in the variable `nbytes` above.

Compile and run the following C program as p3d.c:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
     char buffer[10];
     int n;

     printf("Enter 6 characters\n");
     n = read(0, buffer, 6);

     printf("%d characters were entered\n", n);
     printf("The characters were\n");
     write(1, buffer, n);
}
```

**Open and Close**

Entries in the process's descriptor table are changed as the process opens and closes various I/O objects during its lifetime. For example, the following C program uses the `read` function to read from a file we have created earlier. Notice how the format of the `read` system call is exactly the same, whether the data is coming from the keyboard or from a file. We just supply a device number in fd as the first parameter.

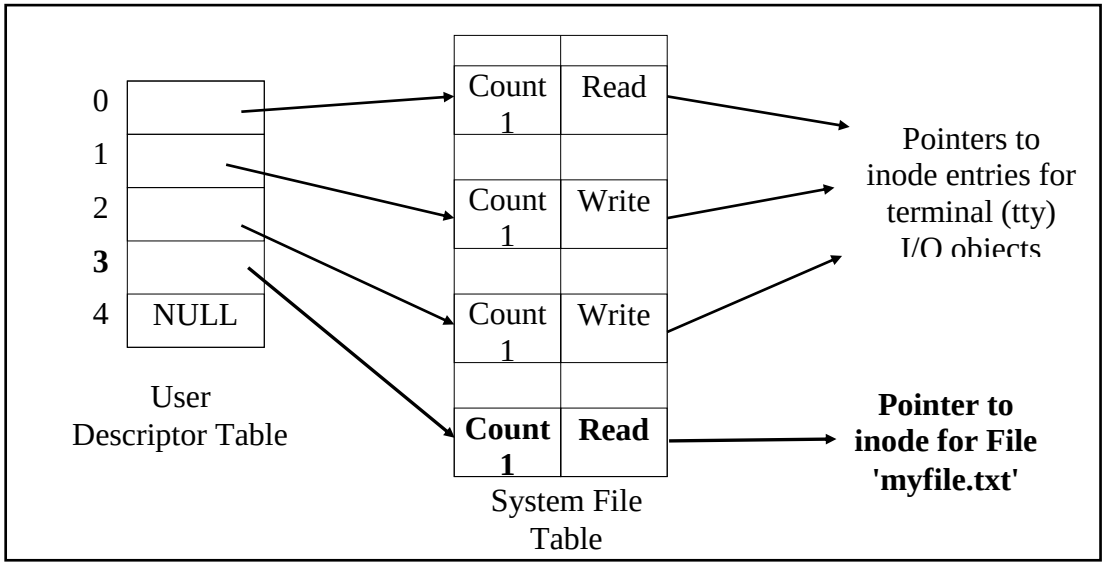
Compile and run this program as p3e.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
      char buffer[10];
      int fd, n;

      fd = open("p3a.c",O_RDONLY);
      n = read(fd, buffer, 6);
      close(fd);

      printf("%d characters were read from file p3a.c\n", n);
      printf("The characters were\n");
      write(1, buffer, n);
      printf("\n");
}
```

**Explanation**

Note we include a header file fcntl.h for file control operations open and close and the constant RD_ONLY.

After the open system call, a new file table entry is created by the system, indicating the file is in use by a single process in read only mode and the reading position is at the start. The next non-NULL entry in the process descriptor table is used to point to the new file table entry. The open system call would return the value 3 to the process in this example. File 'p3a.c' is subsequently accessed as I/O object 3 after the "open" call. As the file is open in read only mode, write operations to descriptor 3 would fail.



When a process is finished with an I/O object, be it a file or a communication object, the `close` system call is invoked to delete the entry from the user descriptor table. The entry is reset to NULL and can later be assigned to point to another I/O object. Furthermore, if no other descriptors in the user tables reference the system's file table entry for that object, then the file table entry will also be deleted which frees system resources and forces any pending I/O operations on the object to be completed before it returns successfully.

**Exercise – Create and compile this exercise as p3f.c**

The `cat` utility in Unix displays the contents of a given file on the text window. Write your own program to do this in C as follows:

The program should ask the user for the name of the file to be printed and read in the user's response using scanf.

The program should then open that file.

The program should then read the entire file of text in blocks of 10 characters and write this to standard output using the byte level read and write functions below.

The program should then close the file and terminate.

Note that when there are no more characters available, the last call to function read will return a count less than 10. You can use a do-while loop which tests for this condition to decide when to terminate.

```
do {
       n=read(fd, buffer, 10); /*Read 10 chars from file*/
       write(1, buffer, n);  /*Write chars to text terminal*/
    } while (n==10); /* keep reading until n < 10 */
```

In order to receive your mark for the lab, please have the 6 C programs ready for inspection by a demonstrator at the end.