Unit VI: Unix System Interface

File Descriptor

File descriptor is integer that uniquely identifies an open file of the process.a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource.Each Unix process has three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	Stdin
1	Standard output	STDOUT_FILENO	Stdout
2	Standard error	STDERR_FILENO	stderr

Low level I/O – read and write, Open, create, close and unlink

Basically there are total 5 types of I/O system calls:

1. Create: Used to Create a new empty file.
Syntax in C language:
int creat(char *filename, mode_t mode)

Parameter:
filename : name of the file which you want to create
mode : indicates permissions of new file.

Returns:
return first unused file descriptor (generally 3 when first creat use in process beacuse 0, 1, 2 fd are reserved)
return -1 when error

Working

- Create new empty file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure\

2. open: Used to Open the file for reading, writing or both.
Syntax in C language
#include<sys/types.h>
#includ<sys/stat.h>
#include <fcntl.h>
int open (const char* Path, int flags [, int mode]);

Parameters
Path : path to file which you want to use absolute path begin with "/", when you are not work in same directory of file.
Use relative path which is only file name with extension, when you are work in same directory of file.

flags : How you like to use
O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write, O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists

Working:

- Find existing file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

Example:

```
// C program to illustrate
// open system call
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
        // if file does not have in directory
        // then file foo.txt is created.
        int fd = open("foo.txt", O_RDONLY | O_CREAT);

        printf("fd = %d/n", fd);

        if (fd ==-1)
        {
                // print which type of error have in a code
                printf("Error Number % d\n", errno);

                // print program detail "Success or failure"
                perror("Program");
        }
        return 0;
}
```

Output:
Fd=3

3. **close:** Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax in C language

```
#include <fcntl.h>
int close(int fd);
```

Parameter
fd :file descriptor
Return
0 on success.
-1 on error.

Working:
Destroy file table entry referenced by element fd of file descriptor table
As long as no other process is pointing to it!
Set element fd of file descriptor table to NULL

Example:

```
// C program to illustrate close system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
        int fd1 = open("foo.txt", O_RDONLY);
        if (fd1 < 0)
        {
                perror("c1");
                exit(1);
        }
        printf("opened the fd = % d\n", fd1);

        // Using close system Call
        if (close(fd1) < 0)
        {
                perror("c1");
                exit(1);
        }
        printf("closed the fd.\n");
}
```

Output:
opened the fd = 3
closed the fd.

4. **read:** From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language
size_t read (int fd, void* buf, size_tcnt);

Parameters
fd: file descripter
buf: buffer to read data from
cnt: length of buffer

Returns: How many bytes were actually read
return Number of bytes read on success
return 0 on reaching end of file
return -1 on error
return -1 on signal interrupt

Important points

- buf needs to point to a valid memory location with length not smaller than the specified size because of overflow.
- fd should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.
- cnt is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read less bytes than cnt.

Example:

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
int fd, sz;
char *c = (char *) calloc(100, sizeof(char));

fd = open("foo.txt", O_RDONLY);
if (fd< 0) { perror("r1"); exit(1); }

sz = read(fd, c, 10);
printf("called read(% d, c, 10). returned that"
                " %d bytes were read.\n", fd, sz);
c[sz] = '\0';
printf("Those bytes are as follows: % s\n", c);
}
```

Output:

```
called read(3, c, 10).  returned that 10 bytes  were read.
Those bytes are as follows: 0 0 0 foo.
```

5. **write:** Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

```
#include <fcntl.h>
size_twrite (int fd, void* buf, size_tcnt);
```

Parameters

fd: file descripter
buf: buffer to write data to
cnt: length of buffer

Returns: How many bytes were actually written
return Number of bytes written on success
return 0 on reaching end of file
return -1 on error
return -1 on signal interrupt

Important points

- The file needs to be opened for write operations

- buf needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.
- cnt is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when fd have a less number of bytes to write than cnt.
- If write() is interrupted by a signal, the effect is one of the following:
- -If write() has not written any data yet, it returns -1 and sets errno to EINTR.
- -If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted

6. **Unlink:** In Unix-like operating systems, unlink is a system call and a command line utility to delete files. The program directly interfaces the system call, which removes the file name and (but not on GNU systems) directories like rm and rmdir.[1] If the file name was the last hard link to the file, the file itself is deleted as soon as no program has it open.

To delete a file named foo:

```
% unlink foo
```

Random access – lseek

Normally, file I/O is sequential: each read or write proceeds from the point in the file right after the previous one. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file, UNIX System V maintains a file-offset that indicates the next byte to be read or written. If n bytes are read or written, the file-offset advances by n bytes. When necessary, however, a file can be read or written in any arbitrary order using lseek to move around in a file without actually reading or writing.

To do random (direct-access) I/O it is only necessary to move the file-offset to the appropriate location in the file with a call to lseek. Calling lseek as follows:

lseek(fildes, offset, whence);
or as follows:
location = lseek(fildes, offset, whence);

forces the current position in the file denoted by file-descriptor fildes to move to position offset as specified by whence. Subsequent reading or writing begins at the new position. The file-offset associated with fildes is moved to a position offset bytes from the beginning of the file, from the current position of the file-offset or from the end of the file, depending on whence; offset may be negative. For some devices (e.g., paper tape and terminals) lseek calls are ignored. The value of location equals the actual offset from the beginning of the file to which the file-offset was moved. The argument offset is of type off_t defined by the header file <types.h> as a long; fildes and whence are int's.
The argument whence can be SEEK_SET, SEEK_CUR or SEEK_END to specify that offset is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append a file, seek to the end before writing:

lseek(fildes, 0L, SEEK_END);
To get back to the beginning (``rewind"),

lseek(fildes, 0L, SEEK_SET);
Notice the 0L argument; it could also be written as (long) 0.

With lseek, you can treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary point in a file:

```
get(fd, p, buf, n) /* read n bytes from position p */
  int fd, n;
  long p;
  char *buf;
{
lseek(fd, p, SEEK_SET);  /* move to p */
  return(read(fd, buf, n));
}
```

Listing Directory in Unix

Display or list all directories in Unix
Type the following command:
$ ls -l | grep `^d'
ls -l | egrep `^d'

Or better try the following ls command only to list directories for the current directory:
$ ls -d */

Programming Method: Debugging

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.

Compiling

To prepare your program for debugging with gdb, you must compile it with the -g flag. So, if your program is in a source file called memsim.c and you want to put the executable in the file memsim, then you would compile with the following command:
gcc -g -o memsimmemsim.c

Invoking and Quitting GDB

To start gdb, just type gdb at the unix prompt. Gdb will give you a prompt that looks like this: (gdb). From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed). Or, you can start gdb and give it the name of the program executable you want to debug by saying gdb executable.To exit the program just type quit at the (gdb) prompt (actually just typing q is good enough).

Commands

1. **Help :** Gdb provides online documentation. Just typing help will gives a list of topics. Then you can type help topic to get information about that topic. Type help command and get information about any other command.

2. **File** :file executable specifies which program you want to debug.
3. **Run :** run will start the program running under gdb.
   You can even do input/output redirection: run > outfile.txt.
4. **Break:** A ``breakpoint'' is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.break function sets the breakpoint at the beginning of function. If your code is in multiple files, you might need to specify filename:function. break linenumber or break filename:linenumber sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.
5. **Delete :** delete will delete all breakpoints that you have set. delete number will delete breakpoint numbered number. You can find out what number each breakpoint is by doing info breakpoints. (The command info can also be used to find out a lot of other stuff. Do help info for more information.)
6. **clear :**clear function will delete the breakpoint set at that function. Similarly for linenumber, filename:function, and filename:linenumber.
7. **Continue :**continue will set the program running again, after you have stopped it at a breakpoint.
8. **Step :** step will go ahead and execute the current source line, and then stop execution again before the next source line.
9. **Next :** next will continue until the next source line in the current function (actually, the current innermost stack frame, to be precise). This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called.
10. **Until :** until is like next, except that if you are at the end of a loop, until will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.
11. **List :** list line_number will print out some lines from the source code around line number. If you give it the argument function it will print out lines from the beginning of that function. Just list without any arguments will print out the lines just after the lines that you printed out with the previous list command.
12. **Print :** print expression will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called list, do print list[0]@25

Macro

You can use macros to simplify formatting and ensure consistency. Macros take advantage of one of the UNIX system's distinguishing characteristics: the capability to build complex processes from basic, primitive units. A macro is nothing more than a series of troff requests, specified and named, that perform special formatting tasks.

User Defined Header

As your programs become larger, and as you start to deal with other people's code (e.g. other C libraries) you will have to deal with code that resides in multiple files.
We saw that one way of including custom-written functions in your C code, is to simply place them in your main source file, above the declaration of the main() function. A better way to re-use functions that you commonly incorporate into your C programs is to place them in their own file, and to include a statement above main() to include that file. When

compiled, it's just like copying and pasting the code above main(), but for the purpose of editing and writing your code, this allows you to keep things in separate files. It also means that if you ever decide to change one of those re-usable functions (for example if you find and fix an error) that you only have to change it in one place, and you don't have to go searching through all of your programs and change each one.

Header files

A common convention in C programs is to write a header file (with .h suffix) for each source file (.c suffix) that you link to your main source code. The logic is that the .c source file contains all of the code and the header file contains the function prototypes, that is, just a declaration of which functions can be found in the source file.

This is done for libraries that are provided by others, sometimes only as compiled binary "blobs" (i.e. you can't look at the source code). Pairing them with plain-text header files allows you see what functions are defined, and what arguments they take (and return).

Example:

Primes.h

```
int isPrime(int n); // returns 0 if n is not prime, 1 if n is prime
```

Primes.c

```
int isPrime(int n) {

  // returns 0 if not prime, 1 if prime

  if (n<2) return 0;         // first prime number is 2
  if (n==2) return 1;        // ensure 2 is identified as a prime
  if ((n % 2)==0) return 0;  // all even numbers above 2 are not prime

  int i;
  for (i=3; i*i< n; i++) {  // test divisibility up to sqrt(n)
    if ((n % i) == 0) {
      return 0;
    }
  }
  return 1;
}
```

Go.c

```
/* go.c
   Takes one input argument from the command line, an integer,
   and returns 1 if the number is prime, and 0 if it is not.

   Compile with: gcc -o go go.cprimes.c
*/

#include <stdio.h>
#include <stdlib.h>
#include "primes.h"

int main(int argc, char *argv[]) {
  if (argc< 2) {
printf("error: must provide a single integer value to test\n");
```

```
    return 1;
  }
  else {
    int n = atoi(argv[1]);
    int prime = isPrime(n);
printf("isPrime(%d) = %d\n", n, prime);
    return 0;
  }
}
```

Output:

```
plg@wildebeest:~/Desktop$ gcc -o go go.cprimes.c
plg@wildebeest:~/Desktop$ ./go 1
isPrime(1) = 0
plg@wildebeest:~/Desktop$ ./go 2
isPrime(2) = 1
plg@wildebeest:~/Desktop$ ./go 3
isPrime(3) = 1
plg@wildebeest:~/Desktop$ ./go 63
isPrime(63) = 0
plg@wildebeest:~/Desktop$ ./go 67
isPrime(67) = 1
plg@wildebeest:~/Desktop$ ./go 12347
isPrime(12347) = 1
```

User Defined Library Function,

A function is a collection of declarations and statements that carries out a specific action and/or returns a value. Previously defined functions that have related functionality or are commonly used (e.g. math or graphics routines) are stored in object code format in library files. Object code format is a special file format that is generated as an intermediate step when an executable program is produced. Like executable files, object code files are also not displayed to the screen or printed. Functions stored in library files are often called library functions or runtime library routines.

The standard location for library file in most Unix system is the directory /usr/lib and /usr/local/lib...

How many types libraries do we have and what is the difference between?

Usually, two basic types of libraries are used in compilations: static libraries and shared object libraries. Static libraries are collection of object files that are unused during the linking phase of a program. Referenced code is extracted from the library and incorporated in the excutable code . Shared libraries contain re-locatable object that can be shared by more than one application. During compilation, the object code from the library is not incorporated in the executable code, but only a reference to the object is made. When the executable that uses a shared object library is loaded into memory the appropriate shared object library is loaded and attached to the image.

The arunility can also be used to create a library. For example, we have 2 functions: ascii which is in ascii.c and change_case which is in change_case.c files.

```
//File: ascii.c
  char * ascii(int start, int finish)
   {
     char *b= new char(finish-start+1);
     for (int i= start; i<=finish; ++i)
         b[i-start]=char(i);
          return b;
    }
//File: change_case.c
  #include<ctype.h>
  char* change_case(char *s)
  {
   char *t = &s[0];
       while(*t)
          {
          if(isalpha(*t))
         *t+=islower(*t)?-32:32;
         ++t;
         }
    return s;
  }
 $ g++ -c change_case.c
 $ g++ -c ascii.c
 $ arcrlibmydemo.aascii.ochange_case.o
```

Compile the code to object code, and the object code added to the archive with the utility ar.

The prototypes for the functions in the mydemo library are placed in a corresponding header file called mydemo.h Be sure to use preprocessor directives in that file to prevent it from being inadvertently included more than once.

```
//File mydemo.h
  #ifndef MYDEMO_H
  #define MYDEMO_H
   char* ascii(int, int);
   char* change_case(char*);
#endif
```

Library file creation
File: main.c

```
#include<iostream>
#include"mydemo.h"
using namespace std;
main()
{
int start, stop;
char b[20];
```

```
cout<<"enter the start and stop value for string:"<<endl;
cin>>start>>stop;
cout<<"create string..."<<ascii(start, stop)<<endl;
cin.ignore(80,'\n');
cout<<"enter a string :";
cin.getline(b,20);
cout<<"coverted string: "<<change_case(b)<<endl;
return 0;
}
```

Next step is to compile and link it:

```
 $ g++ -o main mani.c -L. -lmydemo
```

Here, "-L." is to use to tell the system that when the compiler searches for library files it should also include the current directory. The name of the library is passed using the -l command option, you call the library without the lib prefix and the .aextension..

```
$ main
enter the start and stop value for string:
56 68
create string... 89:,0<1321@ABCD
enter a string : Hello, World.
coverted string: hELLO, wORLD.
```

makefile utility.

There is a UNIX tool called make that is commonly used to compile C programs that are made up of several files, and (sometimes) involve several compilation steps. There is a lot of power in the make tool, but what I want to introduce here is a simple use of it, which lets you avoid having to remember a long, complicated compile command (e.g. in line 1 of the output from the prime number program above).

The make utility uses a special plain-text file that you write that has to reside in the same directory as your program, and has to be called Makefile. You can think of a Makefile as a recipe for making your program (i.e. linking and compiling).

A simple Makefile for our prime number program above might look like this:

```
go: go.cprimes.c
gcc -o go go.cprimes.c
```

The first word and colon go: on line 1 represents the name of a recipe called go. The list of files after the colon (go.cprimes.c) represent all of the things that go depends upon. On the next line, there is a TAB (not spaces) followed by a compile command. This represents the step (there could be more lines for more steps if there were any) that are required to "make" the "go" recipe. Here we simply have put our compile command.

Now all we have to do from the command-line is type make, and make will "make" the recipe for "go". (Running make with no arguments executes the first rule (recipe) in the Makefile). The make program knows that the "go" rule needs to be executed if any of the files that it depends upon, (because they follow the colon on line 1) changes.

Here is what it looks like when we run make using Makefile1.txt:

```
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ cp Makefile1.txt Makefile
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
gcc -o go go.cprimes.c
```

You can see the command (line 3) that ends up being executed by make.

Introducing more generalization to the Makefile

There are a number of features of a Makefile we can utilize to make the whole idea more useful. We can introduce "macros" (like a variable) to generalize the name of the C compiler to use, the flags to pass the compiler, the location of any library files, etc etc. Here is what a more generalized Makefile might look like for our primes example from above:

```
CC = gcc
CFLAGS = -Wall
DEPS = primes.h
OBJ = go.oprimes.o

%.o: %.c $(DEPS)
	$(CC) $(CFLAGS) -c -o $@ $<

go: $(OBJ)
gcc $(CFLAGS) -o $@ $^
```

You can see we have moved all the specific details (filenames, compiler flags, etc) into the macros on the top, and what remains below in the rules themselves, is expressed only in terms of those macros. There's nothing wrong with using a Makefile that is simple (as in Makefile1.txt, it is a choice for you about how fancy to get. The one limitation of Makefile1.txt is that the header file primes.h doesn't appear anywhere … this means if that file changes, then make will not think it has to recompile anything (because nothing in the rule "go" depends on primes.h). In Makefile2.txt, we introduce a dependency of .o files (on line 6) on $(DEPS), which is defined above on line 3, and includes the header file primes.h.

Note that we have term in the CFLAGS macro that looks like this: -Wall. This is a flag to the compiler to turn on all Warnings. There are many warnings that the compiler will tell you about, like variables that are never used, uninitialized variables, etc. Consult the documentation for full details.

Here is what it looks like when we run make using Makefile2.txt:

```
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ cp Makefile2.txt Makefile
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
gcc -c -o go.ogo.c
gcc -c -o primes.oprimes.c
gcc -o go go.oprimes.o
```

You can see that in this case, three commands end up being run (lines 3-5).

Now the neat thing is, if we type make again (without changing anything) we get this:

plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
make: `go' is up to date.
We are told that the "go" rule is "up to date". The make program checks to see which files have changed since the last make, and only executes the step in the rule(s) (if any) that need to be done (it figures this out based on the dependencies that you set up in the rules).

In the long run, using Makefiles is a good idea, because:

it's faster to recompile things (less typing, and it only recompiles based on what's changed and leaves the rest)
it organizes all the "steps" in a (potentially complex) compilation into one place (the Makefile), which makes it easier for other people to compile your code