CS 332/532 – 1G- Systems Programming

Lab 4

Objectives

The objective of this lab is to introduce you to UNIX file I/O functions (`open, read, write, close, lseek`) and the `make` utility.

Most of the lab materials have been taught in the lectures this week. We have included those materials just in case you need them.

Exercise

In this lab you will learn some of the UNIX file I/O functionality such as: `open, read, write, seek` and `close`. We will use an example to learn how to use some of these file I/O functions.

Example #1:

We wrote a C program to copy one file and copy the contents of that file to a new file by performing following steps (here we must provide the input and output file names as *command line arguments.*

You can download the source code: filecopy.c

1. Check if the correct number of arguments is given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three, then the program should print an error message and terminate. Also, input and output file names should not be the same. What happens if you provide the same file name for input and output?

```
if (argc != 3){
  printf("Usage: %s <source> <destination>\n", argv[0]);
  exit (-1);
}
```

2. Use the *open* function in *read only* mode to read the input file. The open function takes the name of the file as the first argument and the open flag as the second argument. The open flag specifies if the file should be opened in read only mode (O_RDONLY), write only mode

(O_WRONLY), or read-write mode (O_RDWR). There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the newly created file. Note that the UNIX file uses {*read, write, execute*} (*rwx)* permissions for the user, group, and everyone. You can find out various modes supported by the open function from the man page or the textbook.

```
readFileDescriptor = open(argv[1], O_RDONLY);
```

The function returns a file descriptor which is typically a non-negative integer. If there is an error opening the file, the function returns -1. Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2. Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, respectively.

3. Use the open function to write the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file. This is accomplished by ORing the different open flags: O_CREAT, O_WRONLY, and O_TRUNC. O_CREAT specifies opening a new empty file if the file does not exist and requires the third file permission argument to be provided. O_WRONLY specifies that the file should be open for writing only and the O_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).

```
writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY| O_TRUNC, 0700);
```

4. Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided. A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error. After you read the data into the buffer write the buffer to the new file using the *write* function. The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the *write* function is successful, it will return the number of bytes written.

```
while ((n=read(readFileDescriptor,buf,BUFFSIZE)) > 0) {
  if (write(writeFileDescriptor, buf, n) != n){
     printf("write error");
  }
}
```

5. After completing the copy process use the *close* function to close both file descriptors. The close function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.

```
close(readFileDescriptor);
close(writeFileDescriptor);
```

Example #2:

The second example was the *lseek* function. We used it to move to particular location in the file and modify it by performing following steps (you can download the code: filelseek.c)

1. Open the output file of Example #1.

2. Use lseek to read last 10 bytes of file and print it on console. The *lseek* function takes three arguments: file descriptor, the file offset, and the base address from which the offset is to be implemented (often referred to as *whence*). In this example, we are trying to read the last 10 bytes in the file, so we set the whence to the end of the file using SEEK_END and specify the offset as 10. You can look at the man page for lseek to find out other predefined whence values: SEEK_SET, SEEK_CUR, etc. The lseek function returns the new file offset on a successful and returns -1 when there is an error.

```
if (lseek(readFileDescriptor, SEEKSIZE, SEEK_END) >= 0){
  if((n=read(readFileDescriptor,buf,BUFFSIZE)) > 0){
    if (write(STDOUT_FILENO, buf, n) != n) {
      printf("write error");
    }
  } else {
   printf("read error");
  }
} else {
  printf("lseek error");
}
```

3. Use lseek to write a string character "THIS IS NEW MSG FROM LSEEK!" at the beginning of the file.

```
if (lseek(RWFileDescriptor, 0, SEEK_SET) >= 0){
  if (write(RWFileDescriptor, lseekMSG, strlen(lseekMSG)) != strlen(lseekMSG)) {
    printf("write error");
  }
} else {
  printf("lseek error (During file writing.)");
}

close(RWFileDescriptor);
```

If you compile and run the above program with a simple input file, you should get the following output:

```
$ cat testfile
This is a test file. This is a test message. Do you see it?
$ ./a.out testfile
u see it?
$ cat testfile
THIS IS NEW MSG FROM LSEEK!s a test message. Do you see it?
$
```

Check out what values are printed if you change the *offset* and *whence* to the following values when you are using the *lseek* function with the *readFileDescriptor*:

offset	whence
0	SEEK_SET
0	SEEK_END
-1	SEEK_END
-10	SEEK_CUR

Make Utility

make is a utility that is used to automatically detect which program needs to be recompiled while working on a large number of source programs and will recompile only those programs that have been modified. The *make* utility uses a *Makefile* to describe the rules for determining the dependencies between the various programs and the compiler and compiler options to use for compiling the programs. In case of C programs, an executable is created from object files (*.o files) and object files are created from source files. Source files are often divided into header files (*.h files) and actual source files (*.c files).

To illustrate the use of make, let us consider adding a new function to measure the time taken by the insertion sort program that we wrote in Lab 2. Instead of adding this method to the same file as the insertion sort, let us create a new file and create a header file that has the method prototype. The two files gettime.h and gettime.c are shown below:

gettime.h:

```
#ifndef _GETTIME_H_
#include <stdio.h>
#include <sys/time.h>
double gettime(void);
#endif
```

gettime.c:

```
#include "gettime.h"

double gettime(void) {
  struct timeval tval;
  gettimeofday(&tval, NULL);
  return((double)tval.tv_sec + (double)tval.tv_usec/1000000.0);
}
```

Note that we can compile the file gettime.c separately and link the object file with any other program that uses the ***gettime*** function. To use the gettime function in the insertion sort program, we have to include the file gettime.h and invoke the ***gettime*** function before and after the call to

insertionsort function. Here are the steps involved in incrementally compiling and linking these two different files:

```
$ gcc –c gettime.c
$ gcc -c insertionsort.c
$ gcc -o insertionsort insertionsort.o gettime.o
```

Also note that we don't have to recompile gettime.c if we are only making changes to the file insertionsort.c. These dependencies are what we can describe in a make file and let the make utility determine which files have been updated and recompile those files.

Here is a simple Makefile to compile and build the above C programs:

```
# Sample Makefile to compile C programs
CC = gcc
CFLAGS = -Wall -g #replace -g with -O when not debugging
DEPS     = gettime.h Makefile
OBJS     = gettime.o insertionsort.o
EXECS    = insertionsort

all:     $(EXECS)

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

insertionsort: $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^

clean:
        /bin/rm -i *.o $(EXECS)
```

Let us review some of the key elements in a Makefile:

- Any text after # is a comment and is ignored by the make utility.
- Constants (they are referred to as macros) are assigned values using the notation: *constant = value1 value2*
- You use these constants in rules we must use $(NAME), where NAME is a constant defined earlier.

- Dependencies are denoted using the notation: *leftside:<tab>rightside* and is followed by what commands to execute in the next line after a tab (it is important that call commands start with a tab, otherwise make will complain about *missing separator*).
- The symbol % is used to denote wildcard entries, e.g., *%.o: %.c* implies that all *.o* files depend on corresponding *.c* files.
- Special macro $@ represents text to the left of : and $^ represents text to the right of : whereas $< represents the first item in text that appears to the right of :

If the Makefile is saved as Makefile or makefile, you can invoke make utility by typing make. If you use a different file name other than Makefile or makefile then you must specify the makefile using the -f option to make. If you type make, you should see the following output:

```
$ make
gcc -Wall -g -c -o gettime.o gettime.c
gcc -Wall -g -c -o insertionsort.o insertionsort.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

If you change gettime.h then you should see all files recompiled and the following output:

```
$ touch gettime.h
$ make
gcc -Wall -g -c -o gettime.o gettime.c
gcc -Wall -g -c -o insertionsort.o insertionsort.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

If you change gettime.c then you should see the following output:

```
$ touch gettime.c
$ make
gcc -Wall -g -c -o gettime.o gettime.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

However, if you only change insertionsort.c you will see the following output:

```
$ touch insertionsort.c
$ make
gcc -Wall -g -c -o insertionsort.o insertionsort.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

If you have not modified any files, if you execute make, you will see that following output:

```
$ make
make: Nothing to be done for 'all'.
```

You can download all the source files and the Makefile from CS332-Lab4.zip and unzip the files.

```
$ cd CS332-Lab04
$ make
```

Task 1

Login to moat.cs.uab.edu, download the source code from the text book (available here: http://www.apuebook.com/src.3e.tar.gz) , and build the examples using make using the following commands:

```
$ mkdir cs332
$ cd CS332
$ mkdir textbook
$ cd textbook
$ wget http://www.apuebook.com/src.3e.tar.gz
$ tar xvfz src.3e.tar.gz
$ cd apue.3e
$ make
$ cd fileio
$ ./seek
```

Lab Assignment #4

Implement a C program that takes two filenames as command-line arguments and concatenates the contents of the second file into the first file. The program should check if the two filenames provided as command-line arguments are the same and print an error message in such a case. After the program executes successfully, the first file should now contain its original content and the contents of the second file. The second file should be unchanged. Here is an example:

```
$ cat file1
Hello World! This is file one!
$ cat file2
Hello World! This is file two!
$ gcc -Wall -o lab4 lab4.c
$./lab4 file1 file2
$ cat file1
Hello World! This is file one!
```

```
Hello World! This is file two!
$ cat file2
Hello World! This is file two!
$
```

Test your program with different file sizes and create a file called README.md and include instructions for compiling and executing your program.

Submission

You are required to submit the lab to Canvas by the deadline. No late submissions will be accepted.

Submission Checklist:

- ❑ Upload the C source file (.c file) to Canvas as part of this lab submission.
- ❑ Make sure to test the program on GitHub codespaces and include a link to your lab in your shared repository on GitHub in the comments.
- ❑ Don't forget your README.md file which should include instructions on compiling, running the program and documentation.

Please do not upload executables or object files. Independent Completion Forms are not required for lab assignments.