Files Operations	**LABORATORY**	**FIVE**
OBJECTIVES		
1. File I/O Library Calls 2. File Descriptors and System Calls 3. I/O Streams		

File I/O Library Calls

Very often, we would use *files* for data input and data output, instead of using the keyboard and the screen. As long as Linux and Unix are concerned, the **file** is a very important concept. Virtually everything is considered to be a file in Unix/Linux, including **pipes** for **I**nter-**P**rocess **C**ommunication in next Lab (for communication between processes on the *same machine*) and **sockets** for networking (for communication between processes on the same or across different machines). To use a file, we need to define a *pointer* to a file of data type **`FILE`**. This pointer of type **`*FILE`** is called a *file handle*. To *open* a file for read/write, use **`fopen`** *library call*. To *close* a file, use **`fclose`** *library call*. Note that a *library call* is a higher level mechanism than a *system call*, providing a more user-friendly interface for a programmer.

To open a file for reading, use "**`r`**" (meaning *read*); e.g., *infilep* = **`fopen`**("**grade.txt**","**`r`**") allows you to read grades from the file "**grade.txt**". To open a file for writing, use "**`w`**" (meaning *write*); e.g., *outfilep* = **`fopen`**("**transcript.txt**","**`w`**") allows you to print outputs to the file "**transcript.txt**". A return value of **NULL** means an error in both library calls. You could read input from a file using **`fscanf`** similar to **`scanf`**. You could send output to a file using **`fprintf`** similar to **`printf`**. There is a *special file handle*, called **`stderr`**, to be used when printing an error message. By default, it will go to screen unless being mapped to elsewhere (e.g., a file). It is a *common practice* in C programs that one would use **`stderr`** for errors. Note that you can detect that there are no more data from the input file, by testing the return value from **`fscanf`** against a special value, **`EOF`** (meaning **E**nd **O**f **F**ile).

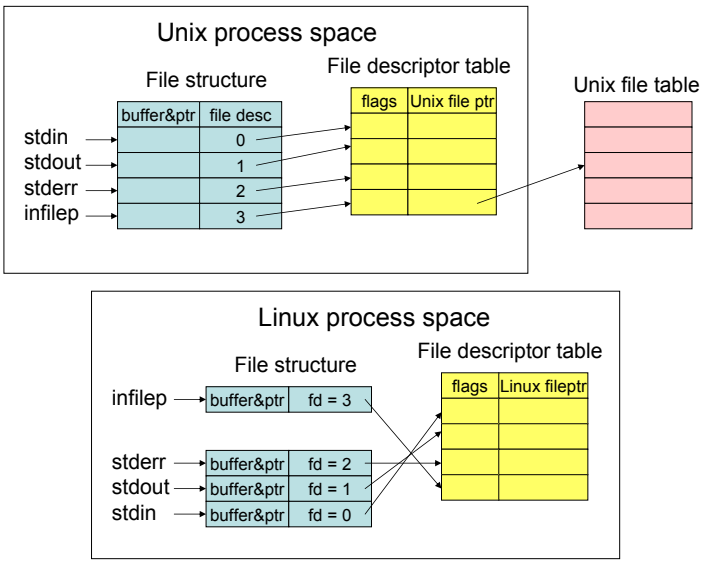
The following program **lab5A.c** is obtained by modifying **lab1C.c** to read the grades from a file and to print the transcript to another file explicitly. Try to run the program in the absence of the input grade file and you will see the error message. Now create the file "**grade.txt**" to store the grades for the student and run it again.

```
// lab 5A
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *infilep, *outfilep;
  int  num_subj, in_gp, sum_gp;
  char in_grade;
  int  err;

  infilep = fopen("grade.txt", "r");
  if (infilep == NULL) {
     printf("Error in opening input file\n");
     exit(1);
  }
  outfilep = fopen("transcript.txt", "w");
  if (outfilep == NULL) {
     printf("Error in opening output file\n");
     exit(1);
  }
  sum_gp = 0;
  num_subj = 0;
  // fscanf will return EOF when end-of-file is reached
  while (fscanf(infilep, "%c\n", &in_grade) != EOF) {
     err = 0;
```

```
      switch (in_grade) {
         case 'A': in_gp = 4; break;
         case 'B': in_gp = 3; break;
         case 'C': in_gp = 2; break;
         case 'D': in_gp = 1; break;
         case 'F': in_gp = 0; break;
         default: fprintf(stderr, "Sorry, wrong grade %c ignored\n", in_grade);
                  err = 1; break;
         // this error message will go to the screen instead of the file
      }
      if (err == 0) {
         num_subj++;
         sum_gp = sum_gp + in_gp;
         fprintf(outfilep, "Grade for subject %d: %c\n", num_subj, in_grade);
      }
   }
   fprintf(outfilep,"Your GPA for %d subjects is %5.2f\n",num_subj,(float)sum_gp/(float)num_subj);
   // close the files when done
   fclose(infilep);
   fclose(outfilep);
}
```

File Descriptors and File System Calls

After opening a file, a *file handle* of type **`FILE*`** is returned. A file handle is a *file pointer* that points to a data structure called a *file structure* in the user area of the process. The file structure contains a *buffer* and a *file descriptor*. This file handle is used in the C *library* **`fscanf`** and **`fprintf`**. A sequence of inputs from a file is a stream of data. Similarly, a sequence of inputs from the keyboard and a sequence of outputs to the screen can also be considered as two *streams* of data. In Unix/Linux, every stream of data is considered as a *file*, which is identified by a *file descriptor*, indirectly via the file handle.



Library calls are more user-friendly, but system calls provide programmers with finer control. A file can be opened with Unix *system call* **`open`**, e.g., *infd* = **`open`**("**grade.txt**", **`O_RDONLY`**), instead of the *library call* **`fopen`**. A *file descriptor* of type **`int`** is returned with **`open`**. It is used as an index into a *file descriptor table* (an array) of the process, storing information about the file, e.g., the *current cursor position* when reading a file. A return value of **-1** means an error when **`open`** is executed (note that **NULL** or **zero** is returned for an error when **`fopen`** is used). To make use of **`open`**, you should include header file **`<fcntl.h>`**. Note that *file handles* are used in **`fscanf`** and **`fprintf`** library calls, but *file descriptors* are used in **`read`** and **`write`** system calls. All library calls to **`fopen`**, **`fscanf`**, **`fprintf`**, and **`fclose`** are finally *translated* into Unix **`open`**, **`read`**, **`write`**, and **`close`** system calls by the C library, which provides a more user-friendly programming interface for a programmer to use. Here is a table for the commonly used *flags* in the **`open`** system call.

open(*pathname*, *open_flag*, *access_mode*)

- More than one of those options in *open_flag* and *access_mode* can be selected, by using "|" (bit-wise *or* operator) to connect them together.
- Note that the *access_mode* argument is needed *only when* a new file is created. Remember to include the appropriate header file **<sys/stat.h>** to use it.

Warning: *do not forget* to provide the third argument on *access_mode* when opening a ***new*** file for writing. Failure to provide that third argument will cause the new file to be created with *strange access privileges*, since the access mode variable will contain *garbage* left behind in the memory space. As a result, it is possible that you could not access your own file, and even worse, anyone could *write to the file* and to implant Trojan horse or virus to your file!

Open flags (inside **fcntl.h**)		**File access modes** (inside **sys/stat.h**)	
O_RDONLY	Read only	**S_IRUSR**	User read
O_WRONLY	Write only	**S_IWUSR**	User write
O_RDWR	Read and write	**S_IXUSR**	User execute
O_APPEND	Append to the end of file on each write	**S_IRGRP**	Group read
		S_IROTH	Other read
O_CREAT	Create the file if not existent	**S_IWGRP**	Group write
		S_IWOTH	Other write
O_TRUNC	If file exists and is opened for write or read/write, truncate the size to zero.	**S_IXGRP**	Group execute
		S_IXOTH	Other execute

Compare the following two programs, with similar behaviors. They make use of the two types of mechanisms for input processing. **Lab5A1.c** is based on **lab5A.c**, but with output file removed to make it easier to compare with **lab5A2.c**. The **read** system call in **lab5A2.c** will place input data in a buffer (an array of characters). User processing logic is needed to handle the input data stored in the buffer.

Lab5A1.c (using fopen in C)

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


int main()
{
  FILE *infilep;
  int  num_subj, in_gp,sum_gp;
  char in_grade;
  int  err;

  infilep = fopen("grade.txt","r");
  if (infilep == NULL) {
     printf("Error in opening input file\n");
     exit(1);
  }
  sum_gp = 0; num_subj = 0;
  while(fscanf(infilep,"%c\n",&in_grade)!=EOF){









      err = 0;
      switch (in_grade) {
        case 'A': in_gp = 4; break;
        ...
      }
      ...
      printf("Grade for ...\n", ...);




  }
  printf("Your GPA ... \n", ...);
  fclose(infilep);
}
```

Lab5A2.c (using open in Unix/Linux)

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

int main()
{
  int  infd;
  int  num_subj, in_gp,sum_gp;
  char inbuf[80];
  int  err, num_char, i;

  infd = open("grade.txt",O_RDONLY);
  if (infd < 0) {
     printf("Error in opening input file\n");
     exit(1);
  }
  sum_gp = 0; num_subj = 0;
  while ((num_char=read(infd,inbuf,80))>0) {
     // zero means EOF, print to debug
     printf("Bytes read = %d\n",num_char);
     i = 0;
     // skip over other characters
     while (i < num_char && ...) i++;
     while (i < num_char) {
        err = 0;
        switch (inbuf[i]) {
          case 'A': in_gp = 4; break;
          ...
        }
        ...
        printf("Grade for ...\n", ...);
        i++;
        while (i < num_char && ...) i++;
      }
  }
  printf("Your GPA ...\n", ...);
  close(infd);
}
```

I/O Streams

In Unix/Linux, when a process reads something from the keyboard, it is actually reading from an *input stream* called **stdin** (standard input). When a process writes something to the screen, it is actually writing to an *output stream* called **stdout** (standard output). The C library call **scanf**(*formatStr*,*&var*) is like **fscanf**(**stdin**,*formatStr*,*&var*) and **printf**(*formatStr*,*var*) is like **fprintf**(**stdout**, *formatStr*,*var*). The third stream **stderr** (standard error) is used in Unix/Linux for printing errors, since it is a good idea to separate special errors from outputs to avoid having them mixed up. By default, **stderr** and **stdout** both refer to the screen, unless a program changes where they should refer to. Like **stderr** mentioned above, the other two standard streams **stdin** and **stdout** can also be considered as file handles.

In Unix/Linux, **stdin** is associated with file descriptor **0**, **stdout** is associated with file descriptor **1** and **stderr** is associated with file descriptor **2**. Try to modify **lab5A2.c** to remove those **open** / **close** calls and replace variable *infd* by **0**. Then you will be reading from the keyboard instead of from the file. This demonstrates that file descriptor **0** is **stdin**. Hit <**Ctrl-D**> to terminate your list of input from the keyboard (<**Ctrl-D**> means *end-of-file* for **stdin**). Note that if you hit <**Ctrl-D**> on the shell, you will automatically exit the shell and log out from the connected terminal or window. To avoid *accidental* hitting of <**Ctrl-D**> in *logging out*, you could set the option variable **ignoreeof** (*ignore EOF*), i.e., **set –o ignoreeof**. To avoid *accidentally overwriting* a file, **set –o noclobber**. To see which option variables have been set currently, type this at the **bash** shell to find out.

```
echo $SHELLOPTS
```

To make your program more *portable*, it is a good practice to use *system constants* to refer to the file descriptors in Unix/Linux for **stdin**, **stdout**, **stderr**, i.e., use file descriptors **STDIN_FILENO**, **STDOUT_FILENO**, **STDERR_FILENO**, instead of simple integers **0**, **1**, **2**. Those Unix/Linux system constants are defined inside <**unistd.h**>. You could replace *infd* by **STDIN_FILENO** in the program and try to run it again.

A call to **open** will return the *smallest integer* corresponding to an *unused* file descriptor. A closed file descriptor will become unused. Try **lab5B1.c** and you will see the *reusing* of file descriptors. Modify the program **lab5B1.c** into **lab5B2.c** by moving the statement **close**(*infd1*) to the end of the program. Now, you will not see the *reusing* effect of the file descriptor, i.e., *infd3* will be **5** instead of **3**.

Note: Although all the programs provided could be compiled under **apollo**, they may generate compilation warnings or errors with respect to system calls like **close** under certain systems using certain compilers. Then you need to include header file <**unistd.h**> explicitly in your programs.

Lab5B1.c

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

int main()
{
  int infd1, infd2, infd3;

  printf("fd 0, 1, 2 are reserved for stdin,
stdout, stderr\n");
  infd1 = open("grade.txt",O_RDONLY);
  infd2 = open("transcript.txt",O_RDONLY);
  printf("fd 1 for grade: %d\n",infd1);
  printf("fd 2 for transcript: %d\n",infd2);
  close(infd1);
  infd3 = open("transcript.txt",O_RDONLY);
  printf("fd 3 for transcript with reusing
closed fd 1: %d\n",infd3);

  close(infd2);
  close(infd3);
}
```

Lab5B2.c

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

int main()
{
  int infd1, infd2, infd3;

  printf("fd 0, 1, 2 are reserved for stdin,
stdout, stderr\n");
  infd1 = open("grade.txt",O_RDONLY);
  infd2 = open("transcript.txt",O_RDONLY);
  printf("fd 1 for grade: %d\n",infd1);
  printf("fd 2 for transcript: %d\n",infd2);

  infd3 = open("transcript.txt",O_RDONLY);
  printf("fd 3 for transcript without reusing
closed fd 1: %d\n",infd3);
  close(infd1);
  close(infd2);
  close(infd3);
}
```

I/O Redirection (Optional)

Remember that a very useful and powerful feature in Unix/Linux is *I/O redirection*. We could get input from a file and send output to a file instead of from the keyboard and to the screen, when running a program that reads input from **stdin** and writes output to **stdout**, by means of

computeGPA < data1.txt > output1.txt

The strength about using I/O redirection is that there is *no change* in the user program **computeGPA.c**, which was originally developed to just *read inputs from the keyboard* and *write outputs to the screen*. As a result, the shell allows you to *glue* different programs together to achieve a possibly complicated task.

If you are to process the command line by yourself (e.g., when you are to implement your own shell), what is the magic behind to allow the input both from the keyboard and from a file to be accessed by the ***same*** executable program **computeGPA** without having to modify the program? That could be achieved by *manipulating the file descriptors*. What we need to do is the make the input file *look like* the keyboard to the program that you develop, i.e., **data1.txt** should look like **stdin**.

We can use the **dup2** system call to manipulate the file descriptors. For example, **dup2**(*fd1*,*fd2*) will first ***close*** *fd2* and then ***copy*** *fd1* to *fd2* so that both *fd1* and *fd2* share the same file pointed to by *fd1*. We could open the input file **data1.txt** as *infd1* and use **dup2**(*infd1*,**0**) to associate it with **stdin**, i.e., replace the keyboard (or use **STDIN_FILENO**). We could also ***open*** the output file **output1.txt** as *outfd2* and use **dup2**(*outfd2*,**1**) to associate that file with **stdout**, i.e., replace the screen (or use **STDOUT_FILENO**). Thus, the solution is to use **dup2** when processing shell command line inputs to replace **stdin** with the file descriptor of **data1.txt** and **computeGPA** will read correctly from the appropriate input stream. Note that there is another similar system call **dup**, but it is ***not*** recommended since the process executing **dup** could be *interrupted* before the return value is assigned and that return value may therefore be corrupted by the interrupting process.

Run the program **lab5C.c** modified from **lab5A1.c** to verify that all output lines sent via **printf** to the screen after the **dup2** system call will go to the file, except for the output line before changing **stdout** via **dup2** and the output lines from **stderr**. You need to include <**sys/stat.h**> in order to use file access mode system constants like **S_IRUSR** and **S_IWUSR**.

```
// lab 5C
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

int main()
{
  FILE *infilep;
  int  num_subj, in_gp,sum_gp;
  char in_grade;
  int  done, outfd;

  infilep = fopen("grade.txt", "r");
  if (infilep == NULL) {
     printf("Error in opening input file\n");
     exit(1);
  }
  printf("First line of output written to screen\n");
  // create the file, if not existent, for write only
  outfd = open("transcript.txt", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
  if (outfd < 0) {
     fprintf(stderr, "Error in writing file\n");
     exit(1);
  }
  fprintf(stderr,"File opened has descriptor %d\n",outfd);
  dup2(outfd,1); // or use dup2(outfd,STDOUT_FILENO);
  fprintf(stderr, "We now see that screen output with fd 1 will be sent to file with fd %d\n",
          outfd);
  printf("Second line of output written to file\n");
  sum_gp = 0;
  num_subj = 0;
```

```
  while (fscanf(infilep, "%c\n", &in_grade) != EOF) {
      switch (in_grade) {
        case 'A': in_gp = 4; break;
        ...
  }
  printf("Your GPA for %d subjects is %f\n", num_subj, (float)sum_gp/num_subj);
  fclose(infilep);
}
```

File Status (Optional)

You could extract information about a file or a directory, by executing the **`stat`** or **`fstat`** system call, found in <**`sys/stat.h`**>. Unlike the difference between **`open`** and **`fopen`** being library call and system call respectively, both **`stat`** and **`fstat`** are system calls. The former takes in the *pathname* of a file and the latter takes in the *file descriptor*. The status information about the file will be returned in a structure. Contents of the returned structure are defined in <**`sys/types.h`**>. Try **lab5D.c** below and check for the status of the different files. Compare the results returned by **`stat`** and **`fstat`**.

```
// lab 5D
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

int main()
{
  char  filename[80];
  int   infd;
  struct stat buf;

  printf("Please enter filename: ");
  scanf("%s", filename);
  printf("File is %s\n", filename);

  // use stat to get some file info
  if (stat(filename, &buf) < 0) {
     printf("Error in stat\n");
     exit(1);
  }
  printf("\nUsing stat system call\n");
  if (S_ISDIR(buf.st_mode))
     printf("%s is a directory of size %d\n", filename, buf.st_size);
  else if (S_ISREG(buf.st_mode))
     printf("%s is a normal file of size %d\n", filename, buf.st_size);
  printf("File owner id %d\n", buf.st_uid);
  printf("File was last accessed at Unix time %d\n", buf.st_atime);
  printf("File was last modified at Unix time %d\n", buf.st_mtime);

  // use fstat to get some file info
  infd = open(filename, O_RDONLY);
  if (infd < 0) {
     printf("Error in opening input file\n");
     exit(1);
  }
  fstat(infd, &buf);
  printf("\nUsing fstat system call\n");
  if (S_ISDIR(buf.st_mode))
     printf("%s is a directory of size %d\n", filename, buf.st_size);
  else if (S_ISREG(buf.st_mode))
     printf("%s is a normal file of size %d\n", filename, buf.st_size);
  printf("File owner id %d\n", buf.st_uid);
  printf("File was last accessed at Unix time %d\n", buf.st_atime);
  printf("File was last modified at Unix time %d\n", buf.st_mtime);
  close(infd);
}
```

Manipulating Directory (Optional)

You can check for the information about a file using program **lab5D.c**. In case of a directory, you could extract the list of files under the directory. To use system calls relevant to directories, you need to include the header file about directory entries <**dirent.h**>. You could then iterate through all the files stored within a directory to display status information about those files under it. To move one more step ahead, you could display all files under all embedded directories *recursively*. Some useful system/library calls to manipulate a *directory* can be found in **lab5E.c**. You could use **opendir** and **closedir** library calls, similar to **fopen** and **fclose** library calls.

```
// lab 5E
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>

int main()
{
  char dirname[80];
  DIR  *dirp;
  struct dirent *direntp;
  struct stat buf;

  printf("Please enter directory name: ");
  scanf("%s", dirname);
  printf("Directory is %s\n", dirname);

  dirp = opendir(dirname);
  if (dirp == NULL) { printf("opendir error\n"); exit(1); }
  // change to that directory
  chdir(dirname);

  // look at each entry in turn
  while ((direntp = readdir(dirp)) != NULL) {
        // stat the file to get more information
        stat(direntp->d_name, &buf);
        // print other information stored inside buf here
       ...
  }
  closedir(dirp);
}
```

Laboratory Exercise (optional)

Making use of your script program **transcript** in **Lab 3**, develop a C version of the program performing the *same functionality* with the *same interface*. In other words, develop a program **transcript.c** which after compilation will produce an executable file **transcript** that will be used in *exactly the same way* as the script program **transcript** in **Lab 3** (so that the user of the program *cannot distinguish* which one they are now using, except that they can find out the file type via the Unix **`file`** command or look inside the file for the magic number). In particular, the program would be run like:

```
transcript COMP* student 1236 1234 1223
```

Here, all files contain the same information in the same format as in **Lab 3**.

Again, the output of your program would look like (and it is easier to format in C):

```
Transcript for 1236 peter
COMP1011 2021 Sem 2 A
COMP2411 2022 Sem 1 A
COMP2432 2022 Sem 2
GPA for 2 subjects 4.00

Transcript for 1234 john
COMP1011 2021 Sem 2 B
COMP2411 2022 Sem 1 B-
GPA for 2 subjects 2.85

Transcript for 1223 bob
COMP1011 2021 Sem 2 F
COMP1011 2022 Sem 1 B
COMP2411 2022 Sem 1 C+
COMP2432 2022 Sem 2
GPA for 2 subjects 2.65
```

In the C program, most of the processing logic in your original script program can be retained, though expressed in different syntax under different data structures. Try to compare the way you program the different functionality for the two versions and observe the difference. This would give you a better understanding on the *relative strength* and *weakness* between writing a script versus a standard program to carry out some simple tasks. For instance, perhaps script is better in handling simple file I/O with text, and standard program is better to handle internal processing logic.

Name your program **transcript.c** and **submit it via BlackBoard on or before 10 March 2023**, to claim *participation score*.