**HW-1B:**

**There are 6 questions in this Homework.**

* **Edit programs 1 and 2 in your Unix/Linux OS and show the outputs that you see along with the source codes.**

**Questions 3 through 6 are also included for Homework 1B**

**Table of content**

[1.Program 1: set status flags (Page 85) 2](#_Toc167295583)

[2.Program 2: Print status flags (Page 84) 5](#_Toc167295584)

[3.Exercise 3.1 10](#_Toc167295585)

[4.Exercise 3.3 12](#_Toc167295586)

[5.Exercise 3.5 12](#_Toc167295587)

[6.Exercise 3.6 16](#_Toc167295588)

[7.Additional HW problem: 19](#_Toc167295589)

Color and symbol specification in Terminal output of this document:

1. gcc hw1\_h.c No shading color: command line;
2. Hello World Yellow shading color: program output;
3. d Blue highlight with yellow shading: (waiting) user input;
4. 🡸 Redirect the STDOUT Red shading: author extra explanation warning
5. $ --- command prompt for NON-root account;
6. # --- command prompt for Root account.

**The program source code and executable files can be found on Github:**

[**https://github.com/Chufeng-Jiang/SFBU-CS510-Advanced-Linux-Programming/tree/main/Homework/Week02**](https://github.com/Chufeng-Jiang/SFBU-CS510-Advanced-Linux-Programming/tree/main/Homework/Week02)

# **1.Program 1: set status flags** (Page 85)

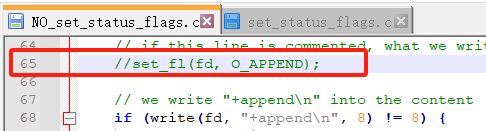
|  |
| --- |
| ***hw1\_set\_status\_flags.c***  This program is to demonstrates how to manipulate file status flags using the fcntl system call to modify the behavior of file descriptors:set\_fl(int fd, int flags), and how the bitwise operations will affect the file status flags.  It first opens (or creates) a file named "*test\_file\_for\_set.txt*" and writes *"Hello, World!"* to it. After closing and reopening the file, the program sets the O\_APPEND flag on the file descriptor using *set\_fl()* function, ensuring that subsequent writes append to the end of the file rather than overwrite existing content. It then appends the string *"+append\n"* to the file, and closes the file again. |
| #include "apue.h"  #include <fcntl.h>  #define BUFFSIZE 8  void set\_fl(int fd, int flags) /\* flags are file status flags to turn on \*/  {  int val;  if ((val = fcntl(fd, F\_GETFL, 0)) < 0)  err\_sys("fcntl F\_GETFL error");  // val = val | flags;  // the result of the bitwise OR operation is assigned back to val.  val |= flags; /\* turn on flags \*/  if (fcntl(fd, F\_SETFL, val) < 0)  err\_sys("fcntl F\_SETFL error");  }  void clr\_fl(int fd, int flags) /\* flags are file status flags to turn on \*/  {  int val;  if ((val = fcntl(fd, F\_GETFL, 0)) < 0)  err\_sys("fcntl F\_GETFL error");  // val = val & ~flags;  // this operation clears the bits in val that are set in flags, while leaving the other bits unchanged. It's used to unset the specific flags.  val &= ~flags; /\* turn on flags \*/  if (fcntl(fd, F\_SETFL, val) < 0)  err\_sys("fcntl F\_SETFL error");  }  int main() {  int fd;    // open the "test\_file\_for\_set.txt",or creat it if it dosen't exist.  // if successfully, we'll get the file descriptor.  if ((fd = open("test\_file\_for\_set.txt", O\_CREAT | O\_WRONLY , 0644)) < 0) {  err\_sys("open error");  }    // using the fd, which points to the "test\_file\_for\_set.txt" file,  // write the "Hello, World!" into the file  if (write(fd, "Hello, World!", 14) != 14) {  err\_sys("write error");  }    // close the fd, and all the content will be saved automatically.  if (close(fd) < 0) {  err\_sys("close error");  }    // we re-open the "test\_file\_for\_set.txt", and get the fd again.  if ((fd = open("test\_file\_for\_set.txt", O\_CREAT | O\_WRONLY , 0644)) < 0) {  err\_sys("open error");  }    // we set the status flag for fd as O\_APPEND.  // it means what we write into the file will be append to the end rather than overwrite from the beginning.  // if this line is commented, what we write will be overwrite the content from the beginning of the file without offset.  set\_fl(fd, O\_APPEND);    // we write "+append\n" into the content  if (write(fd, "+append\n", 8) != 8) {  err\_sys("write error");  }    // close the fd again, and the new content will be saved automatically.  if (close(fd) < 0) {  err\_sys("close error");  }    printf("Test case: test\_file\_for\_set.txt ====> Over.\n");  return 0;  } |

**[Preparation]**

Two c programs were written, one is *hw1\_NO\_set\_status\_flags.c* and another one is *hw1\_set\_status\_flags.c*.

They are almost the same and only with 1 line difference. In the main function of the *hw1\_NO\_set\_status\_flags.c*, the "//*set\_fl(fd, O\_APPEND);* " was commented to avoid the change of the status flag for fd.

Test code and explanations for *hw1\_ NO\_set\_ status\_flags.c* is not included, since it only commented 1 line of code, “//set\_fl(fd, O\_APPEND);” , from *hw1\_ set\_ status\_flags.c*



**[Terminal Coding Tests]**

1. Test in terminal for ***hw1\_NO\_set\_status\_flags.c:***

Without the set\_fl function, in the second time we open the file, what we wrote in to the txt file were always overwrite from the beginning of the content.

|  |  |
| --- | --- |
| $  $  $ | gcc hw1\_NO\_set\_status\_flags.c -o hw1\_no\_set  ./hw1\_no\_set  Test case: test\_file\_for\_NO\_set.txt ====> Over.  cat test\_file\_for\_NO\_set.txt  +append  orld! |

1. Test in terminal for ***hw1\_set\_status\_flags.c***：

With the ***set\_fl()*** function, in the second time we open the file, what we write into the file will be append to the end.

|  |  |
| --- | --- |
| $  $  $ | gcc hw1\_set\_status\_flags.c -o hw1\_set  ./hw1\_set  Test case: test\_file\_for\_set.txt ====> Over.  cat test\_file\_for\_set.txt  Hello, World!+append |

**[Output analysis]**

1. ***hw1\_NO\_set\_status\_flags.c***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H | E | l | l | o | , |  | W | o | r | l | d | ! | \0 |
| After running the program=======> | | | | | | | | | | | | | |
| H + | e a | l p | l p | o e | , n | = d | W\n | o | r | l | d | ! | \0 |

1. ***hw1\_ set\_status\_flags.c***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H | e | L | l | o | , |  | W | o | r | l | d | ! | \0 |  |  |  |  |  |  |  |
| After running the program=======> | | | | | | | | | | | | | | | | | | | | |
| H | e | L | l | o | , |  | W | o | r | l | d | ! | + | a | p | p | e | n | d | \n |

The O\_APPEND flag in the set\_fl() function of file operations is used to open a file in append mode. When a file is opened with this flag, all write operations to the file will automatically move the file offset to the end of the file before writing the data. This ensures that data is always written at the end of the file

# **2.Program 2: Print status** flags (Page 84)

|  |
| --- |
| ***hw2\_print\_status\_flag***  This program is to retrieves and displays the file status flags and access modes associated with a file descriptor provided as a command-line argument. It uses the fcntl() function system call with the F\_GETFL command to obtain the current file status flags. The program then uses bitwise operations and a switch statement to determine and print the file's access mode. Additionally, it checks for and prints the presence of specific flags like O\_APPEND, O\_NONBLOCK, O\_SYNC and O\_FSYNC. |
| #include "apue.h"  #include <fcntl.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  int val;    if (argc != 2)  printf("usage: a.out <descriptor#>");    // get the file descriptor from the sencond position in the cmd line, which is an integer.  // get the file status flags and access modes associated with the file descriptor.  if ((val = fcntl(atoi(argv[1]), F\_GETFL, 0)) < 0)  printf("fcntl error for fd %d", atoi(argv[1]));    // using the switch loop to specify the types of the file status flags.  // O\_ACCMODE defined in the <fcntl.h>, representing the mask for extracting the file access mode from file status flags.  // val & O\_ACCMODE masks out all the bits in val except for those representing the file access mode.  // O\_ACCMODE = 00000011 ***<-- This part will be discussed more in HW2A-Q7***  switch (val & O\_ACCMODE) {  // 00000000 (O\_RDONLY) & 00000011 (O\_ACCMODE) = 00000000  case O\_RDONLY:  printf("read only"); // then it means the file statu is read only.  break;  // 00000001 (O\_WRONLY) & 00000011 (O\_ACCMODE) = 00000001  case O\_WRONLY:  printf("write only");  break;  // 00000010 (O\_RDWR) & 00000011 (O\_ACCMODE) = 00000010  case O\_RDWR:  printf("read write");  break;  default:  printf("unknown access mode");  }    // val & O\_APPEND is another bitwise AND operation.  if (val & O\_APPEND)  printf(", append");  if (val & O\_NONBLOCK)  printf(", nonblocking");  if (val & O\_SYNC)  printf(", synchronous writes");    // This checks if the \_POSIX\_C\_SOURCE, O\_FSYNC macros are not defined  // and if the value of O\_FSYNC is different from the value of O\_SYNC.  // It's ensuring that O\_FSYNC is defined and distinct from O\_SYNC.  #if !defined(\_POSIX\_C\_SOURCE) && defined(O\_FSYNC) && (O\_FSYNC != O\_SYNC)  if (val & O\_FSYNC) // This checks if the file status flags contain the O\_FSYNC flag.  printf(", synchronous writes");  #endif    putchar('\n');  exit(0);  } |

**[Terminal Coding Tests]**

|  |  |
| --- | --- |
| $  $  $  $  $  $  $  $  $  $ | gcc hw2\_print\_status\_flags\_v1.c -o hw2  ./hw2 2 < /dev/tty  read write  ./hw2 1 > temp.foo  $ cat temp.foo  write only  ./hw2 2 2>>temp.foo  write only, append  ./hw2 45555 2>>temp.foo  fcntl error for fd 45555unknown access mode, append, nonblocking, synchronous writes  cat temp.foo  write only  ./hw2 4555 1>>temp.foo  cat temp.foo  write only  fcntl error for fd 4555unknown access mode, append, nonblocking, synchronous writes  ./hw2 5 5<>temp.foo  read write |

**[Output analysis]**

./hw2 0 < /dev/tty

It redirected the STDIN(0) to the file /dev/tty, and pass it to the program, finding its flag status as read only.

./hw2 1 > temp.foo

We redirected the STDOUT(1) to the file temp.foo which is created automatically since it doesn’t exist before, and then we pass STDOUT(1) to the program, finding its flag status as write only.

Since the STDOUT(1) has been directed to temp.foo, the result is not display in the terminal, and it has been written into the temp.foo.

By cat temp.foo, we can see the result.

write only

./hw2 2 2>>temp.foo

The program inspect STDERR(2) and redirects the Standard Error output to the file temp.foo, using append mode. Therefore, by determined the type for the file status, the results is “write only, append”.

>> appends the output to a file, rather than overwriting it.

2>>temp.foo: This redirects the standard error (STDERR) stream (which is file descriptor 2) to append to the file *temp.foo* .

./hw2 45555 2>>temp.foo

I tried to create an error command, by passing the 45555 to the program, which is a number out of bound of the fd range. The terminal displayed the error message, because “2>>temp.foo” has redirect the STDERR(2) to the file *temp.foo*.

* An error message should go to the temp.foo, but I didn’t catch it. (Investigated further)

./hw2 4555 **1**>>temp.foo

I redirect the STDOUT (1) to the file *temp.foo*, therefore the output message will append to the *temp.foo*.

By cat temp.foo, we can see that 1 more line of error message have been appended to the *temp.foo*.

write only

fcntl error for fd 45555unknown access mode, append, nonblocking, synchronous writes

./hw2 5 5<> temp.foo

The textbook has given explanation to the result of this command line saying that “The clause 5<> temp.foo opens the file temp.foo for reading and writing on file descriptor 5.”

<> with a file, it opens the file for both reading and writing

5<>: This opens the file for both reading and writing, associating it with file descriptor 5.

**[Further Investigation of “2>>temp.foo”]**

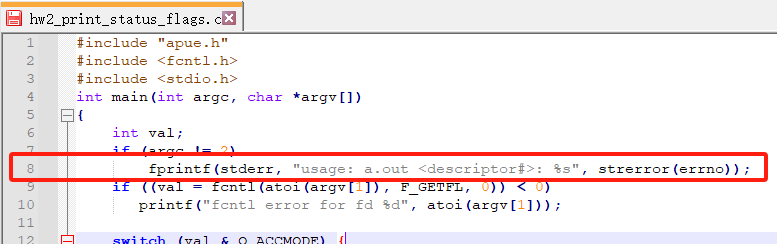
This command means any error messages (stderr) generated by hw2 will be appended to the file temp.foo.

Before we use printf to display the error message, so the result will be given to STDOUT(1), rather than STDERR(2)

Therefore, I modified the source code, and changed the line 8 from *printf* to *fprintf(stderr…)* to output the error message through STDERR(2).

Code modified:

fprintf(stderr, "usage: a.out <descriptor#>: %s", strerror(errno));



And then I re-run the command, a new line of error message was appended to the *temp.foo:*

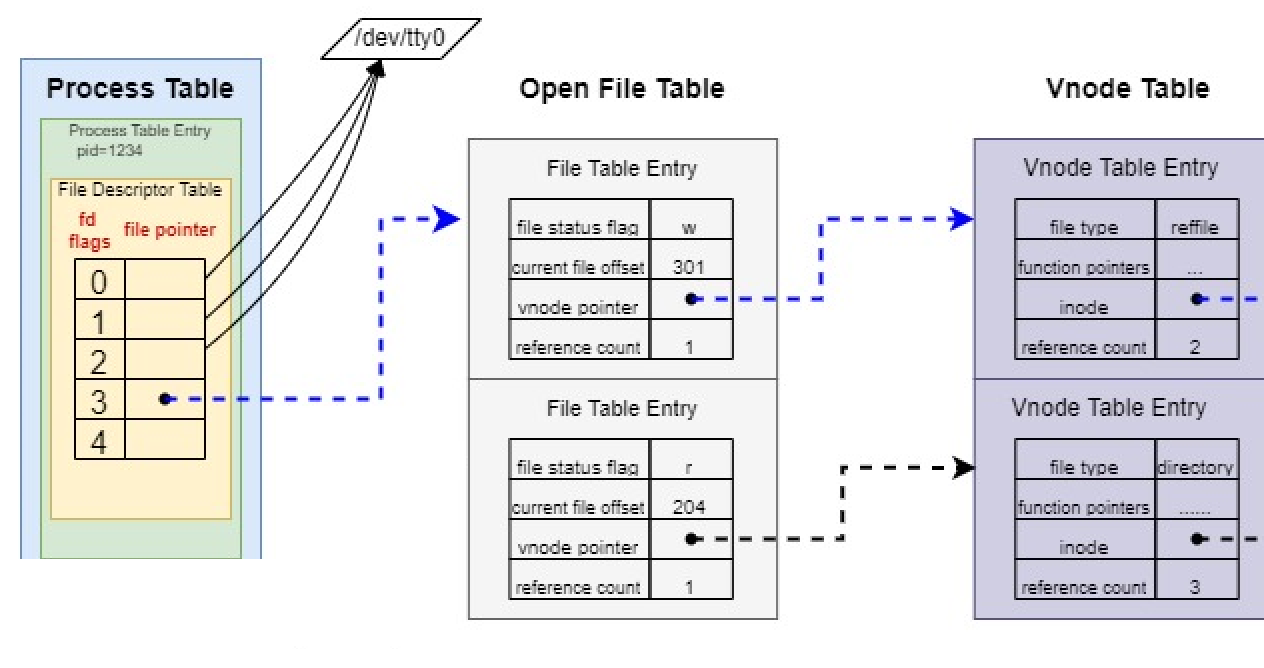
|  |  |
| --- | --- |
| $  $  $ | gcc hw2\_print\_status\_flags.c -o hw2  ./hw2 2>>temp.foo  Segmentation fault (core dumped)  $ cat temp.foo  write only  fcntl error for fd 4555unknown access mode, append, nonblocking, synchronous writes  usage: a.out <descriptor#>: Success 🡨 new error msg appended. |

# **3.Exercise 3.1**

When reading or writing a disk file, are the functions described in this chapter really unbuffered? Explain.

**- 3.1 What is file descriptor?**

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open or create as an argument to either read or write.



**- 3.2 Read/write functions are buffered or not?**

Read/write functions are unbuffered.

The term unbuffered means that each read or write invokes a system call in the kernel. These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Speciﬁcation.

That means each read or write invokes a system call in the kernel, and not buffered in user space programs and libraries or kernel space. The read() and write() functions themselves are direct invocations of a system call and perform I/O directly to or from the kernel, which the file descriptor to read from or write to the associated file or device.

**- 3.3 Function calls**

A function call is an expression containing the function name followed by the function call operator, (). If the function has been defined to receive parameters, the values that are to be sent into the function are listed inside the parentheses of the function call operator. The argument list can contain any number of expressions separated by commas. It can also be empty.

**- 3.4 Automatic buffering for IO**

When a stream is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise, characters may be accumulated and transmitted to or from the host environment as a block.

When a stream is fully buffered, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled.

When a stream is line buffered, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered.

Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment.

In another words, an output stream which is line-buffered shall be flushed whenever a newline is output. An implementation may (but is not required to) flush all line-buffered output streams whenever a read is attempted from any line-buffered input stream. Implementations are not allowed to make streams fully-buffered by default unless it can be determined that they are not associated with an "interactive device".

**Summary**

In conclusion, when reading or writing a disk file, the functions described in this chapter are unbuffered. Since the read() and write() functions are invoking a system call directly and by manipulating the file descriptors, users can read and write the content to the disk file.

# **4.Exercise 3.3**

Assume that a process executes the following three function calls:

fd1 = open(path, oflags);

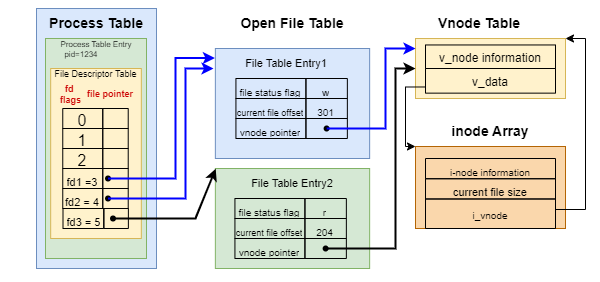
fd2 = dup(fd1);

fd3 = open(path, oflags);

Draw the resulting picture, similar to Figure 3.9. Which descriptors are affected by an fcntl on fd1 with a command of F\_SETFD? Which descriptors are affected by an fcntl on fd1 with a command of F\_SETFL?

|  |
| --- |
| <https://man7.org/linux/man-pages/man2/fcntl.2.html>  fcntl(fd, F\_SETFD, FD\_CLOEXEC);  F\_SETFD (int): Set the file descriptor flags to the value specified by arg (FD\_CLOEXEC).  fcntl(fd, F\_SETFL, flag | O\_NONBLOCK);  F\_SETFL (int): Set the file status flags to the value specified by arg. File access mode (O\_RDONLY, O\_WRONLY, O\_RDWR) and file creation flags (i.e., O\_CREAT, O\_EXCL, O\_NOCTTY, O\_TRUNC) in arg are ignored. On Linux, this command can change only the O\_APPEND, O\_ASYNC, O\_DIRECT, O\_NOATIME, and O\_NONBLOCK flags. It is not possible to change the O\_DSYNC and O\_SYNC flags. |

The operation F\_SETFD on fd1 only modifies the file descriptor flags specific to fd1. However, the operation F\_SETFL on fd1 influences the file table entry that is referenced by both fd1 and fd2.

****

# **5.Exercise 3.5**

The Bourne shell, Bourne-again shell, and Korn shell notation

digit1>&digit2

says to redirect descriptor digit1 to the same file as descriptor digit2. What is the difference

between the two commands shown below? (Hint: The shells process their command lines

from left to right.)

./a.out > outfile 2>&1

./a.out 2>&1 > outfile

**[Preparation]**

1. Write a c program ***hw5\_getc\_hello\_world.c*** using STDOUT and STDERR that can suspend when running.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char c;  // This statement uses STDOUT  printf("Hello, World! \n");  // This statement uses STDERR  perror("This is an error message-->");  // Suspend the program until input from the keyboard  c = getchar();  return 0; } |

**[Terminal Coding Tests]**

1. **Controlled Experiment**

**Test1: - the standard output and redirect of STDOUT**

|  |  |
| --- | --- |
| $  $  $  $ | gcc hw5\_getc\_hello\_world.c -o hw5\_getc  ./hw5\_getc 🡸 No Redirection  Hello, World!  This is an error message-->: Success  d  ./hw5\_getc > hw5\_STDOUT 🡸 Redirect the STDOUT to the file “ hw5\_STDOUT”  This is an error message-->: Success  d  cat hw5\_STDOUT  Hello, World! |

**[Output analysis - 1]**

When we normally running the program without redirection, the STDOUT and STDERR will display their message in the terminal in order, that means the terminal is presenting the outputs as the order of printf() statements when STDOUT and STDER are not redirect to other files.

When we redirect the STDOUT to the file *hw5\_STDOUT*, only the perror message will display in the terminal, while the *printf()* message was write into *hw5\_STDOUT* file.

1. **Test2: ./a.out > outfile 2>&1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Terminal-1:** | | **Terminal-2:** | |
| $  $ | ./hw5\_getc > hw5\_outfile1 2>&1  D  cat hw5\_outfile1  This is an error message-->: Success  Hello, World! | $  $ | ps aux | grep hw5\_getc  beza 2670 0.0 0.0 2776 1408 pts/0 S+ 20:54 0:00 ./hw5\_getc  beza 2674 0.0 0.0 9212 2432 pts/1 S+ 20:55 0:00 grep --color=auto hw5\_getc  ls -l /proc/2670/fd  total 0  lrwx------ 1 beza beza 64 May 18 20:56 0 -> /dev/pts/0  l-wx------ 1 beza beza 64 May 18 20:56 1 -> hw5\_outfile1  l-wx------ 1 beza beza 64 May 18 20:56 2 -> hw5\_outfile1 |

**[Output analysis - 2]**

We run the program in a terminal, nothing displayed on the terminal-1.

We open another terminal, and get the PID for our program. Then we list the process details for the fds. We can find that the fd1 and fd2 are redirected to *hw5\_outfile1*.

After we input a char and the program ended, the *hw5\_outfile1* was automatically created, and in the content the STDERR message goes first than STDOUT because STDERR is unbuffered. After the program ended, the STDOUT message wrote into the *hw5\_outfile1* because STDOUT is line buffered, causing a reverse display order of messages.

1. **Test3: ./a.out 2>&1 > outfile**

|  |  |  |  |
| --- | --- | --- | --- |
| Terminal-1: | | Terminal-2 | |
| $  $ | ./hw5\_getc 2>&1 > hw5\_outfile2  This is an error message-->: Success  d  cat hw5\_outfile2  Hello, World! | $  $ | ps aux | grep hw5\_getc  beza 2707 0.0 0.0 2776 1408 pts/2 S+ 21:37 0:00 ./hw5\_getc  beza 2709 0.0 0.0 9212 2432 pts/1 S+ 21:38 0:00 grep --color=auto hw5\_get  ls -l /proc/2707/fd  total 0  lrwx------ 1 beza beza 64 May 18 21:38 0 -> /dev/pts/2  l-wx------ 1 beza beza 64 May 18 21:38 1 -> hw5\_outfile2  lrwx------ 1 beza beza 64 May 18 21:38 2 -> /dev/pts/2 |

**[Output analysis - 3]**

We run the program in a terminal without any input to prevent the termination of the process. The STDERR message displayed in the terminal-1 immediately.

We open another terminal, and get the PID for our program. Then we list the process details for the fds. We can find that the fd1 is redirected to *hw5\_outfile2*, while fd2 is directed to the terminal.

After we input a char and the program ended, the *hw5\_outfile2* was automatically created, and in the content t **only the STDOUT message is there**.

**[Camparision]**

|  |  |  |
| --- | --- | --- |
| ./hw5\_getc > hw5\_STDOUT | ./a.out > outfile 2>&1 | ./a.out 2>&1 > outfile |
| STDOUT(1) -> *hw5\_STDOUT*  STDERR(2) -> Terminal | STDOUT(1) -> *outfile*  STDERR(2) -> *outfile* | STDOUT(1) -> *outfile*  STDERR(2) -> Terminal |

**[Conclusion]**

./a.out > outfile 2>&1 firstly redirect the STDOUT(1) to the *outfile*, then redirect STDERR(2) to the same file as STDOUT(1), resulting in both of STDOUT(1) and STDERR(2) are redirected to the *outfile*.

./a.out 2>&1 > outfile firstly redirect STDERR(2) to the same file as STDOUT(1) which is the Terminal. Then STDOUT(1) is redirect the *outfile*, resulting in STDOUT(1) redirected to the *outfile* while STDERR(2) is keeping directed to the Terminal.

# **6.Exercise 3.6**

If you open a file for read–write with the append flag, can you still read from anywhere in

the file using lseek? Can you use lseek to replace existing data in the file? Write a program to verify this.

|  |
| --- |
| **hw6\_with\_append.c**  The program demonstrates the impact of the O\_APPEND flag on file descriptor offsets and file size. It performs two tests using different open flags, focusing on O\_APPEND.  Two test has been implemented:  (1)fd = open("hw6\_test\_append.txt",O\_RDWR|O\_APPEND|O\_CREAT|O\_TRUNC,0777);  (1)fd = open("hw6\_test\_append.txt",O\_RDWR| O\_CREAT|O\_TRUNC,0777);  Initially, the program opens or creates the file *hw6\_test\_append.txt* with O\_APPEND flags. It then sets the file cursor to various positions using *lseek()* and writes data to observe how O\_APPEND affects the cursor position. The program sets the cursor to the end of the file, moves it to the 100-byte position, writes 4 bytes, and then moves it to the 70-byte position to write another 8 bytes. Throughout these operations, it prints the cursor's position and the file size (12). Another experiment test is implemented by removing the O\_APPEND flag.  The experiment illustrates that with O\_APPEND, all writes are appended to the file's end, regardless of the cursor position set by *lseek().* |
| #include <sys/stat.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <unistd.h>    int main()  {  int ret=0;  int fd=-1;  struct stat sb;    // open or create a file using append mode.  fd = open("hw6\_test\_append.txt",O\_RDWR|O\_APPEND|O\_CREAT|O\_TRUNC,0777);  if(fd==-1) perror("open() error:");  // set the cursor to the end of the file and get the file size  ret = lseek(fd,0,SEEK\_END);  if(ret==-1) perror("lseek() seek-end error:");  printf("Before lseek(fd,0,SEEK\_END); File Size=%d\n\n",ret);    // set the cursor to the 100 byte position from the begining.  printf("Begin lseek(fd,100,SEEK\_SET)>>>>>>>>\n");  ret=lseek(fd,100,SEEK\_SET);//set offset of file with lseek()  if(ret==-1) perror(">>>>>>>lseek(fd,100,SEEK\_SET) error():");    // get the cursor's current position  ret=lseek(fd,0,SEEK\_CUR);  if(ret==-1) perror("lseek() current cursor error:");  printf("After lseek(fd,100,SEEK\_SET), Current Cursor Posiont = %d\n",ret);  printf("End SEEK\_SET\n\n");  // write 4 byte data into the file  printf("Begin write 4 byte data into file>>>>>>>>\n");  ret=write(fd,"xxxx",4);  if(ret==-1) perror("write() error:");  // get the updated cursor's current position  ret=lseek(fd,0,SEEK\_CUR);  if(ret==-1) perror("lseek() updated current cursor error:");  printf("After write 4 byte data,Current Cursor Posiont = %d\n",ret);    // set the cursor to the 70 byte position from the begining.  printf("Begin lseek(fd,70,SEEK\_SET)>>>>>>>>\n");  ret=lseek(fd,70,SEEK\_SET); //set offset of file with lseek()  if(ret==-1) perror(">>>>>>>lseek(fd,70,SEEK\_SET) error():");    // get the cursor's current position  ret=lseek(fd,0,SEEK\_CUR);  if(ret==-1) perror("lseek() current cursor error:");  printf("After lseek(fd,70,SEEK\_SET), Current Cursor Posiont = %d\n",ret);  printf("End SEEK\_SET\n\n");    // append 8 byte data into the file  printf("Begin write 8 byte data into file>>>>>>>>\n");  ret=write(fd,"12345678",8);  if(ret==-1) perror("write() error:");  // get the updated cursor's current position  ret=lseek(fd,0,SEEK\_CUR);  if(ret==-1) perror("lseek() updated current cursor error:");  printf("After write 4+8 byte data,Current Cursor Posiont = %d\n",ret);  printf("End Write Data\n");    ret=fstat(fd,&sb);  if(ret==-1) perror("fstat error:");  printf("\nFinally, File Size = %ld\n",sb.st\_size);//filesize == 1;    close(fd);  return 0;  } |

**[Terminal Coding Tests]**

|  |  |
| --- | --- |
| $  $ | gcc hw6\_with\_append.c -o hw6\_with  ./hw6\_with  Before lseek(fd,0,SEEK\_END); File Size=0  Begin lseek(fd,100,SEEK\_SET)>>>>>>>>  After lseek(fd,100,SEEK\_SET), Current Cursor Position = 100  End SEEK\_SET  Begin write 4 byte data into file>>>>>>>>  After write 4 byte data, Current Cursor Position = 4  Begin lseek(fd,70,SEEK\_SET)>>>>>>>>  After lseek(fd,70,SEEK\_SET), Current Cursor Position = 70  End SEEK\_SET  Begin write 8 byte data into file>>>>>>>>  After write 4+8 byte data, Current Cursor Position = 12  End Write Data  Finally, File Size = 12 |

**[Output analysis]**

Each time we use lseek(), the cursor will move to the designated position, which means we can read or seek at any position we want.

However, when we use write() function, the cursor will move to the end of the existed content, and concatenate the new data to the end of the file (append action).

**[Further investigation of removing O\_APPEND flag]**

If we modify our source codes, remove the append flag when we open the file, that is

|  |
| --- |
| fd = open("hw6\_test\_append.txt",O\_RDWR|**~~O\_APPEND~~**|O\_CREAT|O\_TRUNC,0777);  fd = open("hw6\_no\_append.txt",O\_RDWR|O\_CREAT|O\_TRUNC,0777); |

**[Terminal Coding Tests]**

|  |  |
| --- | --- |
| $  $ | gcc hw6\_no\_append.c -o hw6\_no  ./hw6\_no  Before lseek(fd,0,SEEK\_END); File Size=0  Begin lseek(fd,100,SEEK\_SET)>>>>>>>>  After lseek(fd,100,SEEK\_SET), Current Cursor Posiont = 100  End SEEK\_SET  Begin write 4 byte data into file>>>>>>>>  After write 4 byte data, Current Cursor Position = 104  Begin lseek(fd,70,SEEK\_SET)>>>>>>>>  After lseek(fd,70,SEEK\_SET), Current Cursor Position = 70  End SEEK\_SET  Begin write 8 byte data into file>>>>>>>>  After write 4+8 byte data, Current Cursor Position = 78  End Write Data  Finally, File Size = **104** |

We can see that without append flag, the cursor will firstly be positioned at 100 bytes, and write 4 bytes of data into it. At this time the file size is 104 bytes.

Then the cursor was re-positioned at 70 bytes, and wrote 8 bytes data into the file which overwrote the existing data there, and doesn’t affect the file size.

In this case, we can *read(), lseek()* and *write()* at any position we want.

# **7.Additional HW problem:**

Homework 1B is posted here. Along with this, also run "type of file" code on Pages 96 and 97 of the textbook.

|  |
| --- |
| ***hw7\_type\_of file.c***  The program is designed to iterate over a list of file names provided as command-line arguments and print the type of each file. The program uses the lstat system call to retrieve the status information for each file and stores it in a struct stat buffer. It then checks the file type using macros provided by the POSIX standard and assigns a corresponding string description to a pointer and print the responding type in the terminal |
| #include"apue.h"  int main (int argc,char \*\*argv)  {  int i;  struct stat buf; // a *“struct stat”* type buffer to store the file’s information  char \*ptr;  // Loop through the command-line arguments, starting from the first file name argument  for(i=1;i<argc;i++)  {  printf("\n%s: ",argv[i]); // print the file name  // get file status information through *lstat()* and store it in buf  if(lstat(argv[i],&buf)<0)  {  err\_ret("lstat error");  continue;  }  // check the file type and set the corresponding description string  if(S\_ISREG(buf.st\_mode))  ptr="regular file";  else if(S\_ISDIR(buf.st\_mode))  ptr="directory file";  else if(S\_ISCHR(buf.st\_mode))  ptr="character special";  else if(S\_ISBLK(buf.st\_mode))  ptr="block special";  else if(S\_ISFIFO(buf.st\_mode))  ptr="fifo ";  else if(S\_ISLNK(buf.st\_mode))  ptr="symbolic link";  else if(S\_ISSOCK(buf.st\_mode))  ptr="socket";  else  ptr="\*\* unknown mode \*\*";  // print out the file type  printf("%s\n",ptr);  }  exit(0);  } |

**[Terminal Coding Tests]**

|  |  |
| --- | --- |
| $  $  > | gcc hw7\_type\_of file.c -o hw7  ./hw7 /etc/passwd /etc /dev/log /dev/tty \  /var/lib/oprofile/opd\_pipe /dev/sr0 /dev/cdrom  /etc/passwd: regular file  /etc: directory file  /dev/log: symbolic link  /dev/tty: character special  /var/lib/oprofile/opd\_pipe: lstat error: No such file or directory  /dev/sr0: block special  /dev/cdrom: symbolic link |

**[Output analysis]**

We can see the output is little different from the text book. It may be resulted by the different system which is working with the program. Since I don’t have the *opd\_pipe* file, so a warning is given in my terminal. Apart from that, other files’ type is correctly printed.