**HW-2A:**

**There are 10 questions in this Homework.**

* **Edit programs in your Unix/Linux OS and show the outputs that you see along with the source codes. There are some \*.txt files associated with some of the source codes. Correlate the \*.txt files with the corresponding source code and run the source codes.**
* **Also provide a brief description of what each code is doing.**

**Table of Content**

[Q1 Sys\_wait.c 2](#_Toc167293656)

[Q2 File\_hole.c 4](#_Toc167293657)

[Q3 File\_open.c 7](#_Toc167293658)

[Q4 File\_read.c 14](#_Toc167293659)

[Q5 File\_write.c 17](#_Toc167293660)

[Q6 File\_close.c 19](#_Toc167293661)

[Q7 File\_flags.c 21](#_Toc167293662)

[Q8 File\_baz.c 27](#_Toc167293663)

[Q9 File\_system\_call.c 32](#_Toc167293664)

[Q10 Printing\_without\_printf.c 34](#_Toc167293665)

[APPENDIX 41](#_Toc167293666)

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.

**All the source code and executable files can be found on the Github:**

[**https://github.com/Chufeng-Jiang/SFBU-CS510-Advanced-Linux-Programming/tree/main/Homework/Week03**](https://github.com/Chufeng-Jiang/SFBU-CS510-Advanced-Linux-Programming/tree/main/Homework/Week03)

# Q1 Sys\_wait.c

|  |
| --- |
| **hw1\_sys\_wait.c**  This program acts as a simple shell and designed a new prompt template “% command” to execute the commands required from the user. It repeatedly prompts the user for input, reads the command entered by the user, and attempts to execute it in a child process. The program removes the trailing newline character from the user input, then uses fork to create a new process. In the child process, it uses *execlp()* to execute the command specified by the user. The parent process waits for the child process to finish using *waitpid()* before prompting the user for another command. |
| #include "apue.h"  #include <sys/wait.h>  int main(void) {  // Declare a buffer array to hold input from the user  char buf[MAXLINE];  pid\_t pid;  int status;  // Print the new designed prompt  printf("%% "); /\* print prompt (printf requires %% to print %) \*/  // Read input from the user, and write it into the buf array  while (fgets(buf, MAXLINE, stdin) != NULL) {  // Check if the last character in the buffer is a newline  if (buf[strlen(buf) - 1] == '\n')  buf[strlen(buf) - 1] = 0; /\* replace newline with null \*/  // Fork a new process and check for errors  if ((pid = fork()) < 0) {  printf("fork error");  } else if (pid == 0) {  // If in the child process, execute the command typed by the user  // int execlp(const char \*file, const char \*arg, ... /\* (char \*) NULL \*/);  // The first buf is the name of the file to be executed. In this context, buf contains the command entered by the user.  // The second buf is the argv[0] value, which conventionally is the name of the program being executed. By convention, this is typically the same as the first argument.  // The (char \*)0 marks the end of the argument list to the new program. In C, (char \*)0 is a null pointer, equivalent to NULL, which signals the end of the arguments.  execlp(buf, buf, (char \*)0);  printf("couldn’t execute: %s\n", buf);  exit(127);  }  // Wait for the child process to terminate  if ((pid = waitpid(pid, &status, 0)) < 0)  printf("waitpid error");  printf("%% ");  }  return 0;  } |

**[Terminal Coding Test]**

|  |  |
| --- | --- |
| Terminal | |
| $  $ | gcc hw1\_sys\_wait.c -o hw1  ./hw1  % date  Mon May 20 06:33:33 PM PDT 2024  % who  beza tty2 2024-05-20 07:48 (tty2)  beza pts/1 2024-05-20 15:55 (192.168.80.1)  beza pts/2 2024-05-20 16:14 (192.168.80.1)  beza pts/3 2024-05-20 16:14 (192.168.80.1)  % ls  apue.h file\_baz foo1.txt hw1 hw10.c hw3\_file\_open.c sysw.c  baz.txt file\_baz.c foo.txt hw10 hw1\_sys\_wait.c sample.txt  % ps  PID TTY TIME CMD  3623 pts/1 00:00:00 bash  4517 pts/1 00:00:00 hw1  4520 pts/1 00:00:00 ps  % ps\ aux  couldn¡¯t execute: ps\ aux  % ps aux  couldn¡¯t execute: ps aux |

**[Output analysis]**

When we execute the program, the “% ” will automatically display in the terminal to indicate us to input our command.

The new prompt is working good will single word command such as “who”, “date”, “ls”, “ps”, however, it cannot work with command with space such as “ps -aux”, “ls -l”.

# Q2 File\_hole.c

|  |
| --- |
| **hw2\_file\_hole.c**  This program creates a file named "*file.hole*", and writes specific content into. Initially, it writes a 10-byte string buf1 *("abcdefghij")* to the file, starting at the beginning. Then, it uses *lseek()* to move the file offset to byte 16384, creating a gap between the first and second write operations. After moving the offset, it writes another 10-byte string buf2 *("ABCDEFGHIJ")* at the new offset. The gap between these writes will result in the creation of holes of “\0”. |
| #include "apue.h"  #include <fcntl.h>  char buf1[] = "abcdefghij";  char buf2[] = "ABCDEFGHIJ";  int main(void) {  int fd;  if ((fd = creat("file.hole", FILE\_MODE)) < 0)  printf("creat error");  // Write 10 bytes from buf1 to the *file.hole*  if (write(fd, buf1, 10) != 10)  printf("buf1 write error");  // The file offset is now 10,  // Move the file offset to 16384 bytes from the beginning of the file, and the file cursor will be at the offset 16384  if (lseek(fd, 16384, SEEK\_SET) == -1)  printf("lseek error");  // Write 10 bytes from buf2 to the file, and the file offset is now 16394.  if (write(fd, buf2, 10) != 10)  printf("buf2 write error");  return(0);  } |

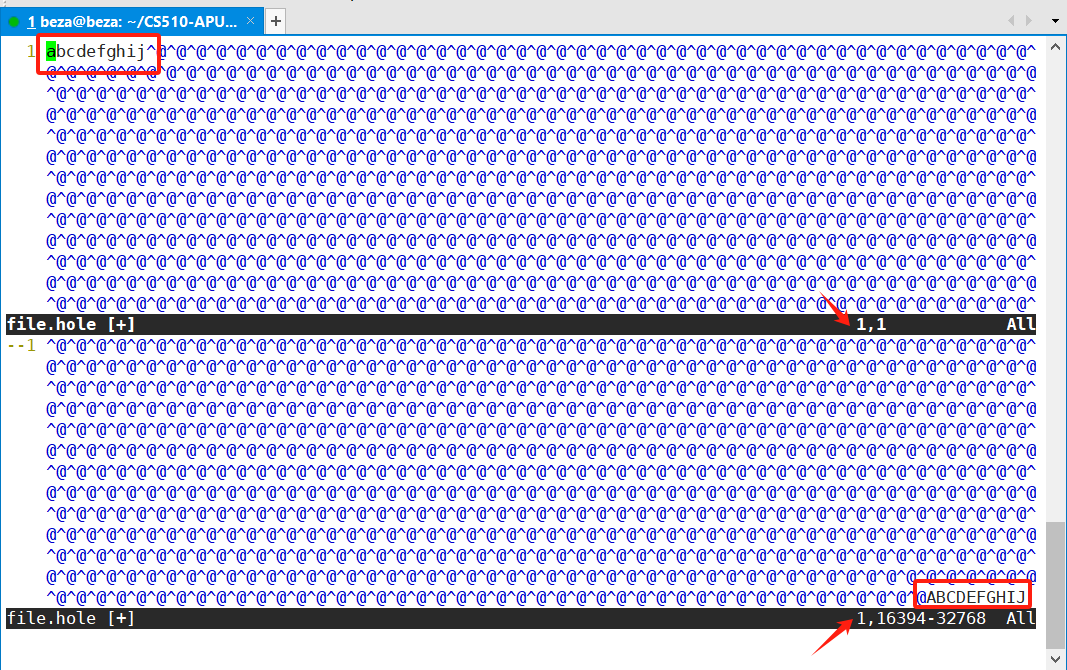
**[Terminal Coding Test]**

|  |  |
| --- | --- |
| Terminal | |
| $  $  $  $  $ | gcc hw2\_file\_hole.c -o hw2  ./hw2  cat file.hole  abcdefghijABCDEFGHIJ  ls -l file.hole  -rw-r--r-- 1 beza beza 16394 May 20 19:49 file.hole  od -c file.hole  0000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0  0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0  \*  0040000 A B C D E F G H I J  0040012 |

**[Output analysis]**

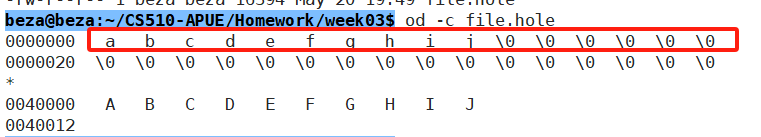
After compiling the code, a *file.hole* was created. When we using cat command, we can see that the contents of the two buffers are connected together “abcdefghijABCDEFGHIJ”.

If we use vim to open the *file.hole, we can see a lot of “^@”.*

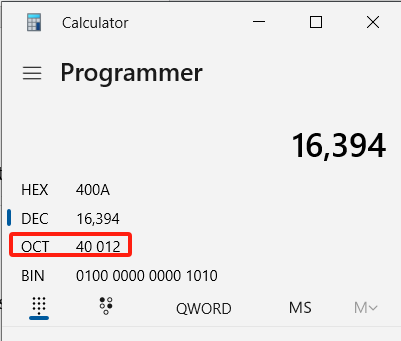


When we use the od(1) command to look at the contents of the file. The -c flag tells it to print the contents as characters. We can see that the unwritten bytes in the middle are read back as zero.

In the terminal, the first number in each line is using octal, Octal (20) = Decimal (16) indicating how many bytes for each line in the terminal, therefore, there are 16 bytes in each line.



The last line number 0040012 indicates the files size in Octal (0040012) = Decimal (16394).



# Q3 File\_open.c

This program is originally to test the file status flag [O\_RDONLY] for a *foo.txt* which is pre-existing in the directory. However, since the code doesn’t contain the access permission bits in the *open()* function, some bugs were found out.

I have rewritten the code, and try to open or create files in different ways. Then I printed out the txt files’ permissions to observe the bugs.

* To better see the bug, we need to delete the created files in advance of running the main program. And no any other operation executed before we get the file descriptors for the “*hw3\_foo\_no.txt*”. We should let the stack keep empty to get some arbitrary bytes. Other wise the mode will be set to a same number every time we execute the program. (Don’t know why and how.)

|  |
| --- |
| **hw3\_file\_open\_no.c**  The program opens files and reads their permission bits, showing how *open()* behaves with different permissions and file existence scenarios.   1. a txt file “*hw3\_foo\_no.txt*” was created, without access permission bits, the file descriptor number and the access permission bits after created were printed; 2. a txt file “*hw3\_foo\_0444.txt*” was created, with access permission bits “0444”, the file descriptor number and the access permission bits after created were printed; 3. a pre-existing txt file “*hw3\_existed\_foo.txt*” was opened, the file descriptor number and the access permission bits after created were printed; 4. a pipe is created to help execute “*ls -l hw3\*.txt*” command to long list the directory file with name starting with “hw3” and end with “.txt”. |
| #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <dirent.h>  // Define the maximum number of lines to get from the terminal  #define MAX\_RESULT\_LENGTH 1000  extern int errno;  int main() {  struct stat file\_stat; // variable to get the file status  int fd1, fd2, fd3;  const char \*command; // command to be executed  FILE \*fp; // file pointer for reading command output  char cmd\_result[MAX\_RESULT\_LENGTH]; // Buffer to store the command result  // open or create a file without specifying access permissions  fd1 = open("hw3\_foo\_no.txt", O\_RDONLY | O\_CREAT);  printf("------Testing O\_CREAT Mode WITHOUT Permission Bit------\n");  printf("Creating File: hw3\_foo\_no.txt ==> fd = %d\n", fd1);  // check if there was an error opening the file  if (fd1 == -1) {  printf("Error Number %d\n", errno);  perror("Program");  }  // get the file status  if (fstat(fd1, &file\_stat) == -1) {  perror("Error in fstat");  close(fd1);  return 1;  }  // print the permission bits of the created file  printf("Creating File: hw3\_foo\_no.txt, Permission Bits: %o\n", file\_stat.st\_mode & (S\_IRWXU | S\_IRWXG | S\_IRWXO));  // create a file with 0444 access permissions  fd2 = open("hw3\_foo\_0444.txt", O\_RDONLY | O\_CREAT, 0444);  printf("\n------Testing O\_CREAT Mode with 0444------\n");  printf("Creating File with 0444: hw3\_foo\_0444.txt ==> fd = %d\n", fd2);  if (fd2 == -1) {  printf("Error Number %d\n", errno);  perror("Program");  }  if (fstat(fd2, &file\_stat) == -1) {  perror("Error in fstat");  close(fd2);  return 1;  }  printf("Creating File: hw3\_foo\_0444.txt, Permission Bits: %o\n", file\_stat.st\_mode & (S\_IRWXU | S\_IRWXG | S\_IRWXO));  printf("\n------Testing O\_CREAT Mode with existing file------\n");  // open an existing file  fd3 = open("hw3\_existed\_foo.txt", O\_RDONLY | O\_CREAT);  printf("Open Existed File: hw3\_foo\_existed.txt ==> fd = %d\n", fd3);  if (fd3 == -1) {  printf("Error Number %d\n", errno);  perror("Program");  }  if (fstat(fd3, &file\_stat) == -1) {  perror("Error in fstat");  close(fd3);  return 1;  }  printf("File: hw3\_foo\_existed.txt, Permission Bits: %o\n", file\_stat.st\_mode & (S\_IRWXU | S\_IRWXG | S\_IRWXO));  printf("\n\n========================================\n");  // command to list files matching the pattern  command = "ls -l hw3\*.txt";  // print the command being executed  printf("Executing command $: %s\n", command);  // execute the command and open a pipe to read the output  fp = popen(command, "r");  if (fp == NULL) {  printf("Failed to execute the command.\n");  return 1;  }  // read the command output line by line  while (fgets(cmd\_result, sizeof(cmd\_result), fp) != NULL) {  // Print each line of the command output  printf("%s", cmd\_result);  }  // close the pipe  pclose(fp);  // close file descriptors  close(fd1);  close(fd2);  close(fd3);  return 0;  } |
| ***hw3\_delete\_files.c***  This program is to delete the txt files created by the above program. |
| #include <fcntl.h>  #include <unistd.h>  #include <stdio.h>  #include <string.h>  #include <dirent.h>  // function to delete files matching specific pattern  void delete\_files() {  DIR \*d; // pointer to directory stream  struct dirent \*dir; // pointer to directory entry  // open the current directory  d = opendir(".");  if (d) {  // read each entry in the directory  while ((dir = readdir(d)) != NULL) {  // check if the file name starts with "hw3\_foo" and ends with ".txt"  if (strncmp(dir->d\_name, "hw3\_foo", 7) == 0 && strstr(dir->d\_name, ".txt")) {  // delete the file  if (unlink(dir->d\_name) == -1) {  perror("unlink");  } else {  // print the name of the deleted file  printf("Deleted: %s\n", dir->d\_name);  }  }  }  // close the directory stream  closedir(d);  } else {  perror("opendir");  }  }  int main() {  delete\_files();  return 0;  } |

## [Terminal Coding Test]

|  |  |
| --- | --- |
| 1. **Terminal** | |
| **$**  **$**  **$**  **$**  **$**  **$**  **$**  **$**  **$**  **$**  **$** | cat hw3\_existed\_foo.txt  hello world  gcc hw3\_file\_open\_no.c -o hw3  ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 130  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 13 May 21 18:07 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:46 hw3\_foo\_0444.txt  ---s-ws--- 1 beza beza 0 May 21 18:46 hw3\_foo\_no.txt  echo "hello world" > hw3\_foo\_no.txt  -bash: hw3\_foo\_no.txt: Permission denied  echo "hello world" > hw3\_foo\_0444.txt  -bash: hw3\_foo\_0444.txt: Permission denied  echo "hello SFBU + 1" > hw3\_existed\_foo.txt  cat hw3\_existed\_foo.txt  hello SFBU + 1  ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 710  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:48 hw3\_foo\_0444.txt  -rwx--s--- 1 beza beza 0 May 21 18:48 hw3\_foo\_no.txt  echo "Bugs with this program?" > hw3\_foo\_no.txt  cat hw3\_foo\_no.txt  Bugs with this program? |

**[Output analysis]**

**Click to go to** [**Appendix A1 - Q3**](#_A1_-_Q3) **to see the full testing output.**

(1) fd1 = open("hw3\_foo\_no.txt", O\_RDONLY | O\_CREAT);

For this way of **creating files**, the access permission code is **randomly** set by the process which can be 130, 710, 570,250,50…

Most of the cases, the file type and permissions for *hw3\_foo\_no.txt* doesn’t contain a write permission, such as permission bit: 130, which is ---s-ws--- in the 1st terminal output above. In this case, if we want to write something in to the file, the terminal will indicate *“-bash: hw3\_foo\_no.txt: Permission denied”.*

There appeared a special circumstances, the file type and permissions for *hw3\_foo\_no.txt is 710,* which is -rwx--s--- in the 2nd terminal output above. In this case, I can write something into this file successfully, even though the read only file status flag was applied for the file descriptor.

By reading the manual, a warning was noticed that can explain the above phenomenon, that each time the permission bit is changing is due to arbitrary bytes from the stack is applied as the file mode(access permission bit).

|  |
| --- |
| <https://man7.org/linux/man-pages/man2/open.2.html>  **int open(const char \****pathname***, int** *flags***, ...** /\* **mode\_t** *mode* \*/ **);**  The mode argument specifies the file mode bits to be applied when a new file is created. If neither O\_CREAT nor O\_TMPFILE is specified in flags, then mode is ignored (and can thus be specified as 0, or simply omitted). The mode argument must be supplied if O\_CREAT or O\_TMPFILE is specified in flags; if it is not supplied, **some arbitrary bytes** from the stack will be applied as the file mode. |

(2)fd2 = open("hw3\_foo\_0444.txt", O\_RDONLY | O\_CREAT, 0444);

With the access permission bit set to 0444, every time when the file was created, it has stable bit of access permission which is 0444, as what we have set in the function.

Under this access permission bit, we cannot write anything into the file.

(3)fd3 = open("hw3\_existed\_foo.txt", O\_RDONLY | O\_CREAT);

We have a pre-existing txt file in our directory in this case. Using this statement, each time we run this program, the file type and permissions for *hw3\_existed\_foo.txt* is *0664,* which is -rw-rw-r--. I can write something into this file successfully, and read the content. And this 0664 is coming from “0666 – umask(0002)”, where 0666 is the default permission code set by the system, however when creating the file the umask is deducted from the default value.

# Q4 File\_read.c

Original code can be found on Appendix-Q4, the content for the file has changed to “hello world@@@@” for better analyzing.

|  |
| --- |
| ***hw4\_file\_read.c***  This program demonstrates file reading and manipulation operations using the *read()* and *lseek()* functions, and how ‘\0’ affect the way of content display in the buffer.  It opens a file named *"hw4\_foo.txt"* in read-only mode and then reads up to 13 bytes from the file into a dynamically allocated buffer. The program prints the current file offset before and after reading, along with the number of bytes read. It then modifies the content of the buffer by appending a character and null-terminating the string at no-continuous position. |
| #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  int main() {  // declare file descriptor and size variables  int fd, sz;    // allocate memory for 100 characters and initialize to 0  char\* c = (char\*) calloc(100, sizeof(char));  // open the file "hw4\_foo.txt" in read-only mode  fd = open("hw4\_foo.txt", O\_RDONLY);  if (fd < 0) {  perror("r1");  exit(1);  }  // Obtain and print the current offset before reading  off\_t offset\_before = lseek(fd, 0, SEEK\_CUR);  printf("Before sz = read(fd, c, 13) : current file offset: %ld\n", (long) offset\_before);  // Read up to 13 bytes from the file into the buffer c  sz = read(fd, c, 13);  // Obtain and print the current offset after reading  off\_t offset\_after = lseek(fd, 0, SEEK\_CUR);  printf("After sz = read(fd, c, 13) : current file offset: %ld\n\n", (long) offset\_after);  // print the number of bytes read from the file  printf("Called read(%d, c, 13). returned that %d bytes were read.\n\n", fd, sz);  printf("Before set c[sz] = \'p\' ===>c[sz] = [%c]\n", c[sz]);  printf("Before set c[sz] = \'p\' ===>Those bytes in buffer c : %s, the length is: %ld\n", c, strlen(c));  // set the character at position sz in buffer c to 'p'  c[sz] = 'p';  printf(" After set c[sz] = \'p\' ===>Those bytes in buffer c : %s, the length is: %ld\n\n", c, strlen(c));  // null-terminate the string read from the file at position sz - 2  c[sz - 2] = '\0';  printf("After set c[sz - 2] = \'\\0\' ===>Those bytes in buffer c : %s, the length is: %ld\n\n ", c, strlen(c));  // free allocated memory and close file descriptor  free(c);  close(fd);  return 0;  } |
| *hw4\_foo.txt:*  hello world@@@@ |

**[Terminal Coding Test]**

|  |  |
| --- | --- |
| Terminal: | |
| $  $ | gcc hw4\_file\_read.c -o hw4  ./hw4  Before sz = read(fd, c, 13) : current file offset: 0  After sz = read(fd, c, 13) : current file offset: 13  Called read(3, c, 13). returned that 13 bytes were read.  Before set c[sz] = 'p' ===>c[sz] = []  Before set c[sz] = 'p' ===>Those bytes in buffer c : hello world@@, the length is: 13  After set c[sz] = 'p' ===>Those bytes in buffer c : hello world@@p, the length is: 14  After set c[sz - 2] = '\0' ===>Those bytes in buffer c : hello world, the length is: 11 |

**[Output analysis]**

The program read 13 bytes to the buffer c, which is a char array created on the heap, because it use the *calloc()* function that allocating 100 memory for an array.

The sz is the return value for the *read()* function, which is the number of bytes read is returned (zero indicates end of file). And sz is always equal to 13.

The index for the 13 bytes should be [0, 12]. When we print the index(13), we can found that nothing is there, because there is an empty space. Then we set the c[13] = p, and print out the buffer, the ‘p’ is printed concatenated at the end of the original 13 bytes, resulting in length increased to 14.

If we set the index(sz -2) to be ‘\0’, which is EOF for a string. When we print the buffer, the string will be ended at the first ‘\0’.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 (sz) | 14 |
| before | h | e | l | l | o |  | w | o | r | l | d | @ | @ | p | \0 |
| after | h | e | l | l | o |  | w | o | r | l | d | \0 | @ | p | \0 |

# Q5 File\_write.c

|  |
| --- |
| ***hw5\_file\_write.c***  This program demonstrates file writing operations using the *write()* function. It first opens a file named "hw5\_foo.txt" with write-only access, creating the file if it doesn't exist and truncating it if it does. The program then writes the string "*hello geeks*" to the file and prints the number of bytes successfully written. Subsequently, it explores whether characters like newline *('\n')* and null terminator *('\0')* affect the length of the string when written to the file. |
| #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  void main()  {  int sz; // declare file descriptor and size variables  // declare file descriptor, the file status flag is with O\_TRUNC, that means:  // (1) if the file is not existed, the create it;  // (2) if the file is already there, open the file and empty the file which means truncating the file and setting the offset to 0.  int fd = open("hw5\_foo.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);  printf("fd = %d\n", fd);  if (fd < 0)  {  perror("r1");  exit(1);  }  // write "*hello geeks* " to the file "*hw5\_foo.txt*", and sz is the return value for write function which means the number of bytes written successfully.  sz = write(fd, "hello geeks", strlen("hello geeks"));  // print the length of "*hello geeks* ", and the number of bytes written into the file.  printf("Called write(%d, \"hello geeks\", strlen(\"hello geeks\") = %ld); ===> It returned %d bytes.\n", fd, strlen("hello geeks"), sz);  // test for the length of "*hello geeks*\n"  printf("\n\n====Does of \"\\n\" counts towards the length of the string? ====\n");  sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));  printf("Called write(%d, \"hello geeks\\n\", strlen(\"hello geeks\\n\") = %ld); ===> It returned %d bytes.\n", fd, strlen("hello geeks\n"), sz);  // test the length of "*hello geeks*\0"  printf("\n\n====Does of \"\\0\" counts towards the length of the string? ====\n");  sz = write(fd, "hello geeks\0", strlen("hello geeks\0"));  printf("Called write(%d, \"hello geeks\\0\", strlen(\"hello geeks\\0\") = %ld); ===> It returned %d bytes.\n", fd, strlen("hello geeks\0"), sz);  close(fd);  } |

**[Terminal Coding Test]**

|  |  |
| --- | --- |
| Terminal: | |
| $  $  $  $  $  $  $ | gcc hw5\_file\_write.c -o hw5  ./hw5  fd = 3  Called write(3, "hello geeks", strlen("hello geeks") = 11); => It returned 11 bytes.  ====Does "\n" counts towards the length of the string? ====  Called write(3, "hello geeks\n", strlen("hello geeks\n") = 12); => It returned 12 bytes.  ====Does "\0" counts towards the length of the string? ====  Called write(3, "hello geeks\0", strlen("hello geeks\0") = 11); => It returned 11 bytes.  cat hw5\_foo.txt  hello geekshello geeks  hello geeks  echo "1111111111111111" > hw5\_foo.txt  cat hw5\_foo.txt  1111111111111111  ./hw5  cat hw5\_foo.txt  hello geekshello geeks  hello geeks |

**[Output analysis]**

A ‘\0’ is always added to the end of a string implicitly, and it doesn’t count towards the length of the string.

A ‘\n’ explicitly write into a string will make a change of line, and this symbol is count towards the length of the string.

When using the O\_TRUNC flag, once the fd open a file successfully, it will remove all the content existed, and write or read the file from the beginning.

# Q6 File\_close.c

|  |
| --- |
| ***hw6\_file\_close.c***  This program demonstrates the use of the *close()* system call to close a file descriptor. It begins by opening an existing file named "*hw6\_foo.txt*" in read-only mode and prints the file descriptor's value. Then, it waits for a keyboard stroke using getc(stdin), to suspend the program and open another terminal to check the file descriptor's details in the process. Finally, it attempts to close the file descriptor, and check the file descriptor's details after the program end. |
| // C program to illustrate close system Call  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main()  {  // open a pre-existing file using read only mode.  int fd1 = open("hw6\_foo.txt", O\_RDONLY);  if (fd1 < 0) {  perror("c1");  exit(1);  }  // print the file descriptor's vale  printf("opened the fd = % d\n", fd1);  // suspend the close of file descriptor until a keyboard stroke happened, to help check the process fds details  int c = getc(stdin);  // close and release the file descriptor, if not close successfully, print “c1”, otherwise print "closed the fd."  if (close(fd1) < 0) {  perror("c1");  exit(1);  }  printf("closed the fd.\n");  } |
| ***hw6\_foo.txt***  hello world |

**[Terminal Coding Test]**

|  |  |  |  |
| --- | --- | --- | --- |
| Terminal 1 | | Terminal 2 | |
| $  $ | gcc hw6\_file\_close.c -o hw6  ./hw6  opened the fd = 3  d  closed the fd. | $     $  $ | ps aux | grep ./hw6  beza 6540 0.0 0.0 2776 1408 pts/0 S+ 07:29 0:00 ./hw6  beza 6639 0.0 0.0 9212 2432 pts/1 R+ 07:30 0:00 grep --color=auto ./hw6  ls -l /proc/6540/fd/  total 0  lrwx------ 1 beza beza 64 May 22 07:32 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 07:32 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 07:32 2 -> /dev/pts/0  lr-x------ 1 beza beza 64 May 22 07:32 3 -> hw6\_foo.txt  ls -l /proc/6540/fd/  ls: cannot access '/proc/6540/fd/': No such file or directory |

**[Output analysis]**

We open two terminal to track the process file descriptors. When the program is running, we could see that the fd(3) is assigned to the program process and point to the pre-existing file “*hw6\_foo.txt*”.

After we input a char “d” and the program ended, we can no longer track the fds for the process. Because the process has been ended, with the file descriptors being closed accordingly.

# Q7 File\_flags.c

|  |
| --- |
| **Copied from HW\_1B\_Q2**  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.  In the HW\_1B\_Q2, I have mentioned that  “// O\_ACCMODE = 00000011 ***<--- This part will be discussed more in HW2A-Q7”***  Here. I am going to investigate more about the file status flags and the file descriptors. |
| #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.  // It's used to check whether the file status flags include the O\_APPEND flag.  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]**

* ***Copied from HW\_1B\_Q2***

|  |  |
| --- | --- |
| $  $  $  $  $  $  $  $  $  $ | 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] *Copied from HW\_1B\_Q2***

./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*.

./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 temp.foo for both reading and writing, associating it with file descriptor 5.

* + **7.1 The File Structure**

The file structure was defined at Line# 991 in the [“/linux/linux/fs.h](https://elixir.bootlin.com/linux/v6.9/source/include/linux/fs.h#L991)” of the source code of the linux kernel version 6.9. More specifically, the file flags were declared at Line# 997.

|  |
| --- |
| <https://elixir.bootlin.com/linux/v6.9/source/include/linux/fs.h#L991>  struct [file](https://elixir.bootlin.com/linux/v6.9/C/ident/file) {  ……  unsigned int [f\_flags](https://elixir.bootlin.com/linux/v6.9/C/ident/f_flags);  …… |

The file flags are declared as unsigned int. In the most architectures, an unsigned integer is a 32-bit (4 bytes) data that encodes a nonnegative integer in the range [0 to 4294967295].

* + **7.2 Flags in *fcntl.h***

In the header file “fcntl.h”, we can find that the flags are defined in macro using 8 bits. And there are total 18 macros defined of which 17 status flags for files and 1 extra O\_ACCMODE.

|  |
| --- |
| #ifndef \_ASM\_GENERIC\_FCNTL\_H  **#define \_ASM\_GENERIC\_FCNTL\_H**  #include <linux/types.h>  /\* open/fcntl - O\_SYNC is only implemented on blocks devices and on files  located on an ext2 file system \*/  **#define O\_ACCMODE 00000003**  **#define O\_RDONLY 00000000**  **#define O\_WRONLY 00000001**  **#define O\_RDWR 00000002**  #ifndef O\_CREAT  **#define O\_CREAT 00000100 /\* not fcntl \*/**  #endif  #ifndef O\_EXCL  **#define O\_EXCL 00000200 /\* not fcntl \*/**  #endif  #ifndef O\_NOCTTY  **#define O\_NOCTTY 00000400 /\* not fcntl \*/**  #endif  #ifndef O\_TRUNC  **#define O\_TRUNC 00001000 /\* not fcntl \*/**  #endif  #ifndef O\_APPEND  **#define O\_APPEND 00002000**  #endif  #ifndef O\_NONBLOCK  **#define O\_NONBLOCK 00004000**  #endif  #ifndef O\_SYNC  **#define O\_SYNC 00010000**  #endif  #ifndef FASYNC  **#define FASYNC 00020000 /\* fcntl, for BSD compatibility \*/**  #endif  #ifndef O\_DIRECT  **#define O\_DIRECT 00040000 /\* direct disk access hint \*/**  #endif  #ifndef O\_LARGEFILE  **#define O\_LARGEFILE 00100000**  #endif  #ifndef O\_DIRECTORY  **#define O\_DIRECTORY 00200000 /\* must be a directory \*/**  #endif  #ifndef O\_NOFOLLOW  **#define O\_NOFOLLOW 00400000 /\* don't follow links \*/**  #endif  #ifndef O\_NOATIME  **#define O\_NOATIME 01000000**  #endif  #ifndef O\_CLOEXEC  **#define O\_CLOEXEC 02000000 /\* set close\_on\_exec \*/**  #endif  #ifndef O\_NDELAY  **#define O\_NDELAY O\_NONBLOCK**  #endif  … |

* + **7.3 Relationship with Flags in *fcntl.h***

The *unsigned int ‘f\_flags’* in the file structure is used to store various flags related to file access and status. This field, typically 32 bits in size, ensures there is enough space to accommodate all possible combinations of these flags. When a file is opened using the *open* system call, the flags specified as arguments (defined in ‘*fcntl.h’*) are stored in the ‘*f\_flags’* field of the file structure. The 32-bit *unsigned int* type of ‘*f\_flags’* allows it to handle all the various 8-bit flags defined in ‘*fcntl.h’*.

* + **7.4 Visualization of File Status Flags**

I run the program from <https://github.com/Cyborg117/File-Descriptors-C>

It can visualize the decimal and bit representation for the status flags well.

|  |  |
| --- | --- |
| Terminal  cd /hw7\_file-descriptors-c/ | |
| $  $ | gcc fcntl\_flags.c -o hw7\_vis  ./hw7\_vis  ===============================================  fcntl.h flags  ===============================================  0\_RDONLY : 0 : 00000000 00000000 00000000 00000000  0\_WRONLY : 1 : 00000000 00000000 00000000 00000001  0\_RDWR : 2 : 00000000 00000000 00000000 00000010  O\_APPEND : 1024 : 00000000 00000000 00000100 00000000  O\_TRUNC : 512 : 00000000 00000000 00000010 00000000  O\_CREAT : 64 : 00000000 00000000 00000000 01000000  O\_WRONLY|O\_APPEND|O\_CREAT: 1089 : 00000000 00000000 00000100 01000001 |

# Q8 File\_baz.c

|  |
| --- |
| **hw8\_file\_baz.c** This program showcases file opening, reading, and closing operations using the *open*(), read(), and *close()* functions. It first opens an existing file named "*hw8\_foo.txt*" in read-only mode, printing the file descriptor assigned to it and then printing out its content character by character. Following a similar pattern, it opens another existing file named "*hw8\_baz.txt*", prints its file descriptor, reads and prints its content, and finally closes its file descriptor.  Throughout the process, the program suspends execution many times and I open another terminal to allow inspection of file descriptor details. This program also points out that “*when a file descriptor (fd) is assigned using the open() system call, it typically returns the lowest available integer that is not currently being used as a file descriptor.*” |
| #include<stdio.h>  #include<fcntl.h>  #include <stdlib.h>  #include <unistd.h>  int main()  {  // declare a char variable to help suspend the program and print the file content.  char ch;  printf("-------open hw8\_foo.txt------\n");  // assume that foo.txt is already created  int fd1 = open("hw8\_foo.txt", O\_RDONLY, 0);  // print the fd assigned to "hw8\_foo.txt".  printf("fd1 = %d\n", fd1);  // print out the content that the fd direct to.  while ((read(fd1, &ch, 1)) != 0) {  putchar(ch);  }  // suspend the program to check the process fds details  ch = getchar();  // close the current fd, to test the fd integer used for the next open() function.  close(fd1);  printf("-------fd1 closed------\n\n");  // suspend the program to check the process fds details  ch = getchar();  printf("-------open hw8\_baz.txt------\n");  // assume that baz.tzt is already created  int fd2 = open("hw8\_baz.txt", O\_RDONLY, 0);  printf("fd2 = %d\n", fd2);  while ((read(fd2, &ch, 1)) != 0) {  putchar(ch);  }  // suspend the program to check the process fds details  ch = getchar();  close(fd2);  printf("-------fd2 closed------\n\n");  // suspend the program to check the process fds details after close the fd.  ch = getchar();  exit(0);  } |
| Pre-existing txt files   1. *hw8\_baz.txt:* Hello, its Baz!   *hw8\_foo.txt:* hello world |

**[Terminal Coding Tests]**

|  |  |  |  |
| --- | --- | --- | --- |
| **Terminal 1** | | **Terminal 2** | |
| $  $ | gcc hw8\_file\_baz.c -o hw8  ./hw8  ---open hw8\_foo.txt---  fd1 = 3  hello world  (press ENTER)  ---fd1 closed---  (press ENTER)  ---open hw8\_baz.txt---  fd2 = 3  Hello, its Baz!  (press ENTER)  ---fd2 closed---  (press ENTER) | $  $  $  $  $  $ | ps aux | grep ./hw8  beza 3241 0.0 0.0 2776 1408 pts/0 S+ 10:25 0:00 ./hw8  beza 3243 0.0 0.0 9212 2432 pts/2 S+ 10:25 0:00 grep --color=auto ./hw8  ls -l /proc/3241/fd  total 0  lrwx------ 1 beza beza 64 May 22 10:25 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 2 -> /dev/pts/0  lr-x------ 1 beza beza 64 May 22 10:25 3 -> hw8\_foo.txt  $ ls -l /proc/3241/fd  total 0  lrwx------ 1 beza beza 64 May 22 10:25 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 2 -> /dev/pts/0  ls -l /proc/3241/fd  total 0  lrwx------ 1 beza beza 64 May 22 10:25 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 2 -> /dev/pts/0  lr-x------ 1 beza beza 64 May 22 10:25 3 -> hw8\_baz.txt  ls -l /proc/3241/fd  total 0  lrwx------ 1 beza beza 64 May 22 10:25 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 10:25 2 -> /dev/pts/0  ls -l /proc/3241/fd  ls: cannot access '/proc/3241/fd': No such file or directory |

**[Output analysis]**

From the output we could know that, when the fd1 open the file *hw8\_foo.txt, fd1(3) ->hw8\_foo.txt,* the system returns the 3 which is the most minimum number in the file descriptor table to the fd1.After, we close fd1, and 3 is unconnected to the file, when checking with the fd files in the system, 3 is no longer there.

We then use fd2 to open the hw8\_baz.txt, and then the system returns the 3 to fd2 again, and fd2(3) *->* hw8\_baz.txt.

That means in the same process, once the fd is closed and released, it will be applied to another variable if it is the current available minimum fd value in the fd table.

After the program ended, the fd table is destroyed, and we can no longer check the status for that process including the fds.

**Q. How file descriptor value is assigned?**

From the Linux kernel source code, the process to find the smallest available file descriptor involves using the *find\_next\_fd*() function, which operates on the fdtable structure.

Initially, the *alloc\_fd*()function [R1] sets the starting point fd to the greater of the given start value or *files->next\_fd*.[R2] It then utilizes the *find\_next\_zero\_bit()* function [R3] to locate the next zero bit in the *full\_fds\_bits* bitmap, starting from the calculated bit position. This determines the next available slot in the bitmap, which is then compared to *maxfd* to ensure it is within bounds. If necessary, start is updated to this new position. Finally, the *find\_next\_zero\_bit()* function searches the *open\_fds* bitmap [R4] from this updated starting point to find the next zero bit, ensuring it identifies the smallest available fd. This systematic search through the bitmap ensures that the function always returns the smallest unused fd.

|  |  |
| --- | --- |
| Reference for the above R1-R4 in the source code. | |
| R4  R1  R2  R3 | <https://elixir.bootlin.com/linux/latest/source/fs/file.c#L499>  //Line 468  struct files\_struct init\_files = {  ……  .fdt = &init\_files.fdtab,  .fdtab = {  .max\_fds = NR\_OPEN\_DEFAULT,  // array of file pointers  .fd = &init\_files.fd\_array[0],  .close\_on\_exec = init\_files.close\_on\_exec\_init,  // a bitmap that contains the file descriptors of currently opened files.  .open\_fds = init\_files.open\_fds\_init,  ……  };  //Line 482  // this function use bitmap to search the opened fds and then get the next unused fd.  static unsigned int find\_next\_fd(struct fdtable \*fdt, unsigned int start)  {  unsigned int maxfd = fdt->max\_fds;  unsigned int maxbit = maxfd / BITS\_PER\_LONG;  unsigned int bitbit = start / BITS\_PER\_LONG;  // this function is defined at “*include/linux/find.h” line 173*  bitbit = find\_next\_zero\_bit(fdt->full\_fds\_bits, maxbit, bitbit) \* BITS\_PER\_LONG;  if (bitbit > maxfd)  return maxfd;  if (bitbit > start)  start = bitbit;  // find the next zero bit in the open\_fds bitmap starting from start, and ***ensures that the smallest available fd is found and returned.***  return find\_next\_zero\_bit(fdt->open\_fds, maxfd, start);  }  //Line 499  static int alloc\_fd(unsigned start, unsigned end, unsigned flags)  {  ……  // Line 509 Search Start  fd = start; // it firstly set it as start value, usually to be 0 because the fd array will be initialized  // if current value lower than the file’s next fd value, set it to be the next fd value  if (fd < files->next\_fd) // if the current fd value lower than the  fd = files->next\_fd;  // if current value is greater then the file’s next fd value, we use the find *find\_next\_fd(fdt, fd);* function to find the next available value.  if (fd < fdt->max\_fds)  fd = find\_next\_fd(fdt, fd);  /\* https://elixir.bootlin.com/linux/latest/source/include/linux/find.h#L173 \*/  //find\_next\_zero\_bit searches for the next zero bit in the open\_fds bitmap, which represents the smallest available fd starting from the given offset. If it finds an available fd, it returns that value.  // find the next zero bit in the open\_fds bitmap starting from start.  unsigned long find\_next\_zero\_bit(const unsigned long \*addr, unsigned long size,  unsigned long offset)  {  if (small\_const\_nbits(size)) {  unsigned long val;  if (unlikely(offset >= size))  return size;  val = \*addr | ~GENMASK(size - 1, offset);  return val == ~0UL ? size : ffz(val);  }  return \_find\_next\_zero\_bit(addr, size, offset);  } |

# Q9 File\_system\_call.c

|  |
| --- |
| **hw9\_file\_sys\_call.c**  This program demonstrates how file descriptors maintain ***separate file offsets*** even when they reference the same file. It opens the file "*hw9\_sample.txt*" twice, resulting in two different file descriptors (fd1 and fd2).  The program reads one byte from each file descriptor and prints the byte and the current file offset for each. It then reads another byte using fd1 and shows that the file offset for fd2 remains unchanged, highlighting that each file descriptor has its own independent offset. This illustrates the isolation of file offsets in concurrent file operations. |
| #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main()  {  char c[10], ch;  // to record the fd offset after each read.  off\_t position1, position2;  // using fd1 and fd2 open the same file  printf("==Read 1 byte for fd1 and fd2==\n");  int fd1 = open("hw9\_sample.txt", O\_RDONLY, 0);  int fd2 = open("hw9\_sample.txt", O\_RDONLY, 0);  // read 1 byte of data to buffer c, and print it.  read(fd1, &c, 1);  printf("fd1: c = %s\n", c);  // get the current offset for fd1  position1 = lseek(fd1, 0, SEEK\_CUR);  if (position1 == -1) {  perror("lseek");  exit(EXIT\_FAILURE);  }  printf("fd1: current position1: %ld\n\n", (long)position1);  // read 1 byte of data to buffer c, and print it.  read(fd2, &c, 1);  printf("fd2: c = %s\n", c);  // get the current offset for fd2  position2 = lseek(fd2, 0, SEEK\_CUR);  if (position2 == -1) {  perror("lseek");  exit(EXIT\_FAILURE);  }  printf("fd2: current position2: %ld\n\n", (long)position2);  // suspend the program to check the fds details  ch = getchar();  // using fd1, read 1 byte of data to buffer c, and print it.  // but fd2 keep no change.  printf("==Read 1 more byte for fd1, NOT fd2==\n");  read(fd1, &c, 1);  printf("fd1: c = %s\n", c);  position1 = lseek(fd1, 0, SEEK\_CUR);  if (position1 == -1) {  perror("lseek");  exit(EXIT\_FAILURE);  }  printf("fd1: current position1: %ld\n", (long)position1);  printf("fd2: current position2: %ld\n", (long)position2);  close(fd1);  close(fd2);  exit(0);  } |
| Pre-existing *hw9\_sample.txt*: This is a sample file. |

**[Terminal Coding Tests]**

|  |  |  |  |
| --- | --- | --- | --- |
| **Terminal 1** | | | **Terminal 2** |
| $  $ | gcc hw9\_file\_sys\_call.c -o hw9  ./hw9  ==Read 1 byte for fd1 and fd2==  fd1: c = T  fd1: current position1: 1  fd2: c = T  fd2: current position2: 1  d  ==Read 1 more byte for fd1, not fd2==  fd1: c = h  fd1: current position1: 2  fd2: current position2: 1 | $ | ls -l /proc/3334/fd  total 0  lrwx------ 1 beza beza 64 May 22 11:29 0 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 11:29 1 -> /dev/pts/0  lrwx------ 1 beza beza 64 May 22 11:29 2 -> /dev/pts/0  lr-x------ 1 beza beza 64 May 22 11:29 3 -> hw9\_sample.txt  lr-x------ 1 beza beza 64 May 22 11:29 4 -> hw9\_sample.txt |

**[Output analysis]**

When we using different file descriptor to open() then same file, different value will assigned to the different file descriptors.

Apart from that, the fds direct to the same file, what they read from the content are working separately. Thant means the offset for the fds after read are moving individually, and they don’t affect each other.

# Q10 Printing\_without\_printf.c

|  |
| --- |
| ***hw10\_print\_wo\_printf.c***  This program demonstrates file operations and cursor position handling using two file descriptors and buffers.  It opens two files, writes "hello world" to the first file, and prints the buffer contents and cursor positions for both file descriptors.  It then reads 12 bytes from the first file into a buffer and writes the buffer content to standard output (STDOUT), effectively displaying the content of the first file.  Cursor positions for both file descriptors are displayed again, showing that operations on one file descriptor do not affect the other. And the *read()* and *write()* function are sharing the same fd offset when using the same fd. |
| #include <fcntl.h>  #include <stdio.h>  #include <string.h>  #include <unistd.h>  #include <stdlib.h>  int main(void)  {  int fd[2]; // using a integer array to store the value for 2 file descriptors.  off\_t cursorPos1, cursorPos2; // to record the offset for the fds  FILE \*file; // a pointer that points to a file, to help print the content for a file  char character;  char buf1[12] = "hello world"; // buffer1 with initialized content.  char buf2[12];  // assume foo1.txt and sample.txt are already created  fd[0] = open("hw10\_foo.txt", O\_RDWR);  fd[1] = open("hw10\_sample.txt", O\_RDWR);  if (fd[0] == -1 || fd[1] == -1) {  perror("Error opening files");  exit(1);  }  printf("===1. after write(fd[0], buf1, strlen(buf1))====");  // writing [strlen(buf1)] bytes of the content from bu1 to file fd[0].  write(fd[0], buf1, strlen(buf1));  // print the content in buf1 and buf2  printf("\nbuf1: %s", buf1);  printf("\nbuf2: %s", buf2);  // get the offset of fd[0] after the write function  cursorPos1 = lseek(fd[0], 0, SEEK\_CUR);  if (cursorPos1 == -1) {  perror("Error getting cursor position");  close(fd[0]);  close(fd[1]);  exit(1);  }  // get the offset of fd[1] after the write function  printf("\nfd[0]: Current cursor1 position: %ld\n", (long)cursorPos1);  cursorPos2 = lseek(fd[1], 0, SEEK\_CUR);  if (cursorPos2 == -1) {  perror("Error getting cursor position");  close(fd[0]);  close(fd[1]);  exit(1);  }  printf("fd[1]: Current cursor2 position: %ld\n", (long)cursorPos2);  printf("\n>>>>>> Print File content in fd[0]: hw10\_foo.txt<<<<<<\n");  // open “*hw10\_foo.txt*” in read mode  file = fopen("hw10\_foo.txt", "r");  if (file == NULL) {  perror("Error opening file");  return 1;  }  // print the content in “*hw10\_foo.txt*” of which the file descriptor is fd[0]  while ((character = fgetc(file)) != EOF) {  printf("%c", character);  }  // close the file  fclose(file);  printf("\n\n\n===2. write(1, buf2, read(fd[0], buf2, 12))====\n");  /\*  This function is write the content in buffer2 to the file descriptor 1(STDOUT);  However, the third parameter is a read() function, and the return value is the number of bytes read from fd[0] to the buffer2. Therefore, is successfully read, the third parameter would be 12.  In summary, this statement was executed as the following sequence:   1. *read(fd[0], buf2, 12)):* read 12 bytes of date from the file directed by fd[0] , and put the content to the buf2; 2. write(1, buf2, 12)): write 12 bytes of date from buf2, them direct to the file descriptor 1 (STDOUT). 3. In this case, the terminal STDOUT will display the content.   fd[0] ---read----*>"hw10\_foo.txt" read(fd[0], buf2, 12))*  |  |----(write)----->buf2----(write)-- > STDOUT *write(1, buf2, read(fd[0], buf2, 12))*  in this way, *write(1, buf2, read(fd[0], buf2, 12))* is equal to *printf()* .  \*/  write(1, buf2, read(fd[0], buf2, 12));  printf("\nbuf1: %s", buf1);  printf("\nbuf2: %s", buf2);  cursorPos1 = lseek(fd[0], 0, SEEK\_CUR);  if (cursorPos1 == -1) {  perror("Error getting cursor position");  close(fd[0]);  close(fd[1]);  exit(1);  }  // Print the current cursor position  printf("\nfd[0]: current cursor1 position: %ld\n", (long)cursorPos1);  cursorPos2 = lseek(fd[1], 0, SEEK\_CUR);  if (cursorPos2 == -1) {  perror("Error getting cursor position");  close(fd[0]);  close(fd[1]);  exit(1);  }  // Print the current cursor position  printf("fd[1]: current cursor2 position: %ld\n", (long)cursorPos2);  close(fd[0]);  close(fd[1]);  return 0;  } |
| Pre-existing files:  *hw10\_foo.txt*: Hello, how are you.  *hw10\_sample.txt*: This is a sample |

**[Terminal Coding Tests]**

|  |  |
| --- | --- |
| Terminal | |
| $  $ | gcc hw10\_print\_wo\_printf.c -o hw10\_wo  ./hw10\_wo  ===1. after write(fd[0], buf1, strlen(buf1))====  buf1: hello world  buf2:  fd[0]: Current cursor1 position: 11  fd[1]: Current cursor2 position: 0  >>>>>> Print File content in fd[0]: hw10\_foo.txt<<<<<<  hello worldare you.  ===2. write(1, buf2, read(fd[0], buf2, 12))====  are you.  buf1: hello world  buf2: are you.  fd[0]: current cursor1 position: 19  fd[1]: current cursor2 position: 0 |

**[Output analysis]**

The original buf1, buf2 , and the file content in foo.txt is as follows:

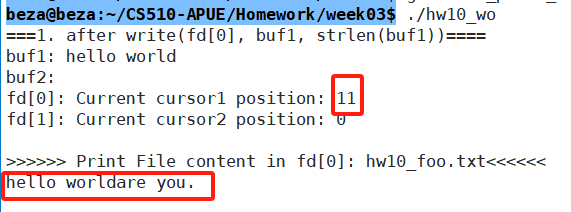
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| buf1 | h | e | l | l | o |  | w | o | r | l | d | \0 |  |  |  |  |  |  |  |  |
| buf2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| fd[0] | H | e | l | l | o | , |  | h | o | w |  | a | r | e |  | y | o | u | . | \0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

After the program execute the 1st write statement: write(fd[0], buf1, strlen(buf1))

buf1, buf2 , the file content in foo.txt, and the offset of the fd[0] is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| buf1 | h | e | l | l | o |  | w | o | r | l | d | \0 |  |  |  |  |  |  |  |  |
| buf2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| fd[0] | h | e | l | l | o | , |  | h | o | w |  | a | r | e |  | y | o | u | . | \0 |
| after write(fd[0], buf1, strlen(buf1)) | | | | | | | | | | | | | | | | | | | | |
| fd[0] | h | e | l | l | o |  | w | o | r | l | d | a | r | e |  | y | o | u | . | \0 |
| ⬆ fd[0] offset\_1 | | | | | | | | | | | | | | | | | | | | |

Here we got the output in the terminal as what we expected:



Then the program reach the statement of *write(1, buf2, read(fd[0], buf2, 12)),* we split this statement into 2 steps to analyze:

1. *read(fd[0], buf2, 12)):* read 12 bytes of date from the file directed by fd[0] , and put the content to the buf2;
2. *write(1, buf2, 12)):* write 12 bytes of date from buf2, them direct to the file descriptor 1 (STDOUT).

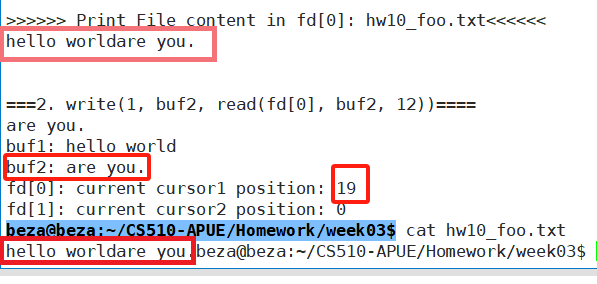
**Step1:** *read(fd[0], buf2, 12)):*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| buf1 | h | e | l | l | o |  | w | o | r | l | d | \0 |  |  |  |  |  |  |  |  |
| buf2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| fd[0] | h | e | l | l | o |  | w | o | r | l | d | a | r | e |  | y | o | u | . | \0 |
| ⬆ fd[0] offset\_1 | | | | | | | | | | | | | | | | | | | | |
| *read(fd[0], buf2, 12)):* | | | | | | | | | | | | | | | | | | | |  |
| fd[0] | h | e | l | l | o |  | w | o | r | l | d | a | r | e |  | y | o | u | . | \0 |
| ⬆ fd[0] offset\_1 fd[0] offset\_2 ⬆ | | | | | | | | | | | | | | | | | | | | |
| buf2 | a | r | e |  | y | o | u | . |  |  |  |  |  |  |  |  |  |  |  |  |

**Step2:** *write(1, buf2, 12)):*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| buf1 | h | e | l | l | o |  | w | o | r | l | d | \0 |  |  |  |  |  |  |  |  |
| buf2 | a | r | e |  | y | o | u | . | \0 |  |  |  |  |  |  |  |  |  |  |  |
| fd[0] | h | e | l | l | o |  | w | o | r | l | d | a | r | e |  | y | o | u | . | \0 |
| fd[0] offset\_2 ⬆ | | | | | | | | | | | | | | | | | | | | |
| *write(1, buf2, 12)):* | | | | | | | | | | | | | | | | | | | |  |
| STDOUT | a | r | e |  | y | o | u | . |  |  |  |  |  |  |  |  |  |  |  |  |

After the second write() operation statement of *write(1, buf2, read(fd[0], buf2, 12)),* we also got the results as expected.



By revieving the source code for the file structure at the “fs.h” Line# 1008, we can find that the structure of file contains a filed of f\_pos for each file, and the f\_pos is a pointer to the current file position. It keeps track of where the next read operation should start.:

|  |
| --- |
| <https://elixir.bootlin.com/linux/v6.9/source/include/linux/fs.h#L1008>  struct file {  ……  // a pointer to the current file position.  // it keeps track of where the next read operation should start.  loff\_t f\_pos; // <---- current offset is defined here  unsigned int f\_flags;  ……} |

In the book “*Linux Device Drivers*”, a more detailed explanation is there. It says that:

|  |
| --- |
| <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch03s04.html>  “loff\_t f\_pos;  The current reading or writing position. loff\_t is a 64-bit value (long long in gcc terminology). The driver can read this value if it needs to know the current position in the file, but should never change it (read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f\_pos directly).” |

By reviewing the Linux manual page for *read()* and write*()*, we could have a better understanding of how the offset effect the *read()* and *write()* function resulting in the content change in the “*hw10\_foo.txt*”. Whether *read()*or *write()* function, as long as they share the same file descriptor, then they share the same file offset information.

|  |
| --- |
| <https://man7.org/linux/man-pages/man2/read.2.html>  ssize\_t read(int *fd*, void *buf* [.*count*], size\_t *count* );  DESCRIPTION  read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.  RETURN VALUE  On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested…… |

|  |
| --- |
| <https://man7.org/linux/man-pages/man2/write.2.html>  ssize\_t write(int *fd*, const void *buf* [.*count*], size\_t *count*);  DESCRIPTION  write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd* ……  For a seekable file (i.e., one to which [lseek(2)](https://man7.org/linux/man-pages/man2/lseek.2.html) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written…… |

# APPENDIX

## A1 - Q3 Original Code:

|  |
| --- |
| **Original file\_open.c** |
| #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <unistd.h>  extern int errno;  int main()  {  // if file does not have in directory  // then foo.txt is created.  // open or created the file of which the type is ready only.  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;  } |
| The content in pre-existing foo.txt: hello world |

## A1 - Q3 Full Terminal Outputs

[**Click here to return to Q3 Terminal Output**](#_[Terminal_Coding_Test])

|  |
| --- |
| $ cat hw3\_existed\_foo.txt  hello world  $ gcc hw3\_file\_open\_no.c -o hw3  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 570  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 13 May 21 18:07 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:46 hw3\_foo\_0444.txt  -r-xrwx--T 1 beza beza 0 May 21 18:46 hw3\_foo\_no.txt  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 130  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 13 May 21 18:07 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:46 hw3\_foo\_0444.txt  ---s-ws--- 1 beza beza 0 May 21 18:46 hw3\_foo\_no.txt  $ echo "hello world" > hw3\_foo\_no.txt  -bash: hw3\_foo\_no.txt: Permission denied  $ echo "hello world" > hw3\_foo\_0444.txt  -bash: hw3\_foo\_0444.txt: Permission denied  $ echo "hello SFBU + 1" > hw3\_existed\_foo.txt  $ cat hw3\_existed\_foo.txt  hello SFBU + 1  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 150  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:47 hw3\_foo\_0444.txt  ---xr-x--T 1 beza beza 0 May 21 18:47 hw3\_foo\_no.txt  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 170  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:48 hw3\_foo\_0444.txt  ---srws--- 1 beza beza 0 May 21 18:48 hw3\_foo\_no.txt  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 250  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:48 hw3\_foo\_0444.txt  --w-r-x--- 1 beza beza 0 May 21 18:48 hw3\_foo\_no.txt  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 50  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:48 hw3\_foo\_0444.txt  ---Sr-x--T 1 beza beza 0 May 21 18:48 hw3\_foo\_no.txt  $ ./hw3\_del  Deleted: hw3\_foo\_no.txt  Deleted: hw3\_foo\_0444.txt  $ ./hw3  ------Testing O\_CREAT Mode WITHOUT Permission Bit------  Creating File: hw3\_foo\_no.txt ==> fd = 3  Creating File: hw3\_foo\_no.txt, Permission Bits: 710  ------Testing O\_CREAT Mode with 0444------  Creating File with 0444: hw3\_foo\_0444.txt ==> fd = 4  Creating File: hw3\_foo\_0444.txt, Permission Bits: 444  ------Testing O\_CREAT Mode with existing file------  Open Existed File: hw3\_foo\_existed.txt ==> fd = 5  File: hw3\_foo\_existed.txt, Permission Bits: 664  ========================================  Executing command $: ls -l hw3\*.txt  -rw-rw-r-- 1 beza beza 15 May 21 18:47 hw3\_existed\_foo.txt  -r--r--r-- 1 beza beza 0 May 21 18:48 hw3\_foo\_0444.txt  -rwx--s--- 1 beza beza 0 May 21 18:48 hw3\_foo\_no.txt  $ echo "Bugs with this program?" > hw3\_foo\_no.txt  $ cat hw3\_foo\_no.txt  Bugs with this program? |

[**Click here to return to Q3 Terminal Output**](#_[Terminal_Coding_Test])

## A2 - Q4 Original Code

|  |
| --- |
| // C program to illustrate  // read system Call  #include <fcntl.h>  #include <stdio.h>  #include <unistd.h>  int main()  {  int fd, sz;  char\* c = (char\*) calloc(100, sizeof(char));  fd = open("hw4\_foo.txt", O\_RDONLY);  if (fd < 0) {  perror("r1");  exit(1);  }  sz = read(fd, c, 13); //  printf("called read(% d, c, 13). returned that %d bytes were read.\n", fd, sz);  c[sz] = '\0';  printf("Those bytes are as follows: % s\n", c);  return 0;  }  **Foo.txt**  hello world@@@@ |

**Q5 Original Code**

|  |
| --- |
| // C program to illustrate  // write system Call  #include<stdio.h>  #include <fcntl.h>  main()  {  int sz;  int fd = open("foo.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);  printf("fd = %d\n", fd);  if (fd < 0)  {  perror("r1");  exit(1);  }  sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));  printf("called write(%d, \"hello geeks\\n\", %d)."  " It returned %d\n", fd, strlen("hello geeks\n"), sz);  close(fd);  }  **Foo1.txt**  hello geeks  ~ |

## A3 - Q8 Original Code

|  |
| --- |
| // C program to illustrate close system Call  #include<stdio.h>  #include<fcntl.h>  int main()  {  // assume that foo.txt is already created  int fd1 = open("hw8\_foo.txt", O\_RDONLY, 0);  close(fd1);  // assume that baz.tzt is already created  int fd2 = open("hw8\_baz.txt", O\_RDONLY, 0);  printf("fd2 = % d\n", fd2);  exit(0);  }  ~ **Foo.txt**  hello world  **Baz.txt**  Hello, its Baz!  **~** |

## A4 - Q9 Original Code

|  |
| --- |
| #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main()  {  char c;  int fd1 = open("sample.txt", O\_RDONLY, 0);  int fd2 = open("sample.txt", O\_RDONLY, 0);  read(fd1, &c, 1);  printf("c = %c\n", c);  read(fd2, &c, 1);  printf("c = %c\n", c);  exit(0);  }  Output:  **Sample.txt**  This is a sample file. |

## A5 - Q10 Original Code

|  |
| --- |
| // C program to illustrate  // I/O system Calls  #include <fcntl.h>  #include <stdio.h>  #include <string.h>  #include <unistd.h>  int main(void)  {  int fd[2];  char buf1[12] = "hello world";  char buf2[12];  // assume foobar.txt is already created  fd[0] = open("foo1.txt", O\_RDWR);  fd[1] = open("sample.txt", O\_RDWR);  write(fd[0], buf1, strlen(buf1));  write(1, buf2, read(fd[1], buf2, 12));  close(fd[0]);  close(fd[1]);  return 0;  }  ~  foo1.txt: Hello, how are you.  sample.txt : This is a sample. |