CSE344 HOMEWORK 1 REPORT

PART 1

C program is designed to append data to a file using the write() system call, with the option to use either O_APPEND or lseek() before each write, depending on the presence of the third command-line argument (x). The program creates the specified file if it doesn't exist already and writes the specified number of bytes to the file by writing one byte at a time. When the O_APPEND flag is used, the write() system call automatically appends the data to the end of the file. In contrast, when lseek() is used, the program first seeks to the end of the file using lseek(fd, 0, SEEK_END) and then writes the data, which overwrites any data written to the file by other processes that may have run concurrently.

```
bahadiretka@Bahadrs-MacBook-Air systemodev % gcc -o part1 part1.c
bahadiretka@Bahadrs-MacBook-Air systemodev % ./part1 f1 1000000 & ./part1 f1 1000000
[1] 15738
bahadiretka@Bahadrs-MacBook-Air systemodev %
[1]  + done       ./part1 f1 1000000
bahadiretka@Bahadrs-MacBook-Air systemodev % ./part1 f2 1000000 x & ./part1 f2 1000000 x
[1] 15753
bahadiretka@Bahadrs-MacBook-Air systemodev %
[1]  + done       ./part1 f2 1000000 x
bahadiretka@Bahadrs-MacBook-Air systemodev % ls -l f1 f2
-rw-r--r--  1 bahadiretka  staff  2000000 Mar 30 22:09 f1
-rw-r--r--  1 bahadiretka  staff  1002841 Mar 30 22:09 f2
bahadiretka@Bahadrs-MacBook-Air systemodev %
```

The file f1 is 2 MB in size, while f2 is only 1 MB. This is because when the program is run without the "x" argument, the O_APPEND flag is used to ensure that all writes are performed atomically at the end of the file. This means that the two instances of the program do not overwrite each other's writes, and the final size of the file is twice the number of bytes specified in the command-line argument. However, when the program is run with the "x" argument, the O_APPEND flag is omitted, and instead an lseek() call is performed before each write. This means that the two instances of the program can overwrite each other's writes, resulting in a final file size of only the number of bytes specified in the command-line argument.

PART 2

The code is an implementation of the dup() and dup2() system calls in C, which are used to duplicate file descriptors. dup() takes an existing file descriptor, and returns a new descriptor that refers to the same file or socket. It does this by calling the fcntl() function with the F_DUPFD argument, which returns the lowest available file descriptor greater than or equal to the specified value. If successful, the new file descriptor is returned. If not, dup() returns -1 to indicate an error. dup2() is similar to dup(), but allows the caller to specify the new file descriptor. If the new descriptor is already in use, dup2() closes it before reusing it. If oldfd is equal to newfd, dup2() checks if oldfd is a valid descriptor, and returns newfd without doing anything else. If oldfd is not valid, it sets errno to EBADF and returns -1. Both functions make use of the fcntl() function, which is used to manipulate file descriptors. fcntl() is called with the F_GETFL flag to check if a file descriptor is open, and with the F_DUPFD flag to create a new file descriptor that refers to the same file or socket as an existing one. The close() function is used to close existing file descriptors that may be in use. In summary, these implementations use the fcntl() and close() functions to duplicate file descriptors in a way that is similar to the dup() and dup2() system calls. They provide an efficient way to create copies of file descriptors, allowing multiple processes to access the same file or socket.

```
bahadiretka@Bahadrs-MacBook-Air systemodev % ./part2
fd1=3, fd2=4, fd3=5
bahadir etka kilinc
bahadiretka@Bahadrs-MacBook-Air systemodev %
```

PART 3

C program demonstrates how duplicated file descriptors share a file offset value and open file. It starts by opening a file "testfile.txt" in read-only mode using the open() system call, which returns a file descriptor fd1. It then duplicates this file descriptor using the dup() system call, which returns a new file descriptor fd2 that refers to the same open file description as fd1. The program then reads from both file descriptors using the read() system call, which reads up to a certain number of bytes from a file into a buffer. It prints the data that was read to the console. Finally, it uses the lseek() system call to get the file offset value for both file descriptors and prints them to the console. The file offset value is the position in the file where the next read or write operation will occur. By comparing the file offset values of both file descriptors, the program demonstrates that they share the same file offset. This is because both file descriptors refer to the same open file description, so any change in the file offset made by one file descriptor will be reflected in the other. The purpose of this program is to illustrate the concept of duplicated file descriptors and how they share a file offset value and open file.

```
[bahadiretka@Bahadrs-MacBook-Air systemodev % gcc -o part3 part3.c
[bahadiretka@Bahadrs-MacBook-Air systemodev % ./part3
Buffer 1: bahadir-etka
Buffer 2: -kilinc
File offset 1: 19
File offset 2: 19
Inode number 1: 30207865
Inode number 2: 30207865
bahadiretka@Bahadrs-MacBook-Air systemodev %
```