7376 HW6

Name: Xuebao Zhao NUID: 002108354

Problem1:

(a) The program first opens a file named "file1.txt" using the “open()” system call with the flags O_CREAT, O_WRONLY, S_IRUSR, and S_IWUSR. These flags specify that if the file doesn't exist, create it, write-only mode, and set the file permission to read and write for the user who runs the program. If the “open()” system call fails, the program prints an error message and exits.

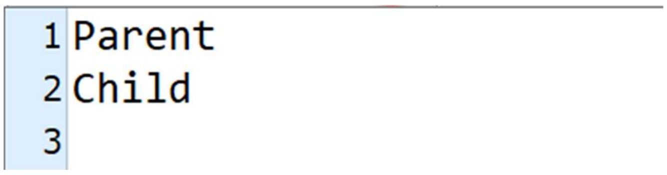
After opening the file, the program calls the “fork()” system call to create a new process. If “fork()” returns a negative value, it means that the fork failed. In this case, the program prints an error message and exits. If “fork()” returns zero, it means that the current process is the child process, and the program writes the string "Child\n" to the file descriptor using the “write()” system call. After the loop, the child process prints a message "Write a character from child".

If “fork()” returns a positive value, it means that the current process is the parent process, and the program writes the string "Parent\n" to the file descriptor using the “write()” system call. After the loop, the parent process prints a message "Write a character from parent".

Finally, the program returns 0, indicating successful execution. When the program is run, the output will show the messages printed by the parent and the child processes alternately every second. The file "file1.txt" will contain the strings "Parent\n" and "Child\n" written in alternating order.

Why they can both access the same physical file:
Both the parent and child can access the same physical file using the descriptor returned by “open()” because the file descriptor is a reference to an open file description, which is maintained by the operating system. When a process opens a file, the operating system creates an open file description for that file, which includes information such as the file offset and file status flags. When a process forks, the child process inherits a copy of the parent process's open file descriptors, including the file descriptor for the open file. Both the parent and child processes have their own stack and data segments, but they share the same open file description and can access the same physical file using the file descriptor.

Result:



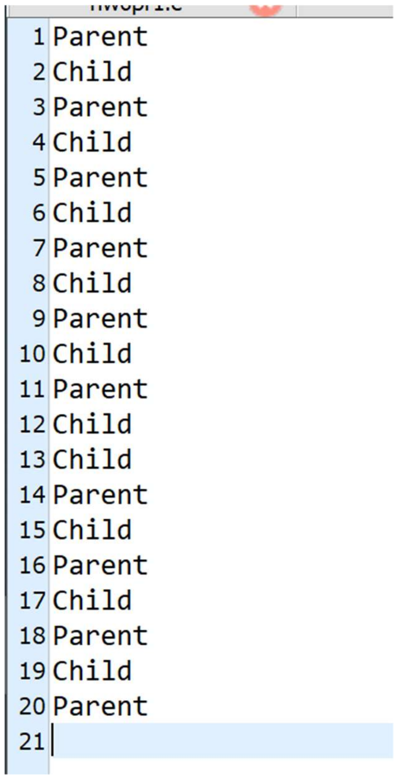
```
Parent
Child

```

(b) Change the code in (a), call “write()” system call in a loop 10 times with 1-second delay between each write in both parent and child.

When both the parent and the child are writing to the file concurrently, the file content can become garbled or corrupted. This is because both processes can be writing to the same file offset simultaneously, leading to data being overwritten or lost.

Result:



```
Parent
Child
Parent
Child
Parent
Child
Parent
Child
Parent
Child
Parent
Child
Child
Parent
Child
Parent
Child
Parent
Child
Parent
```

Problem2:

info: This action takes a filename as argument and prints the inode number, size, and permissions of the file using the “stat()” system call.

link: This action takes two filenames as arguments and creates a hard link from the second filename to the first filename using the “link()” system call.

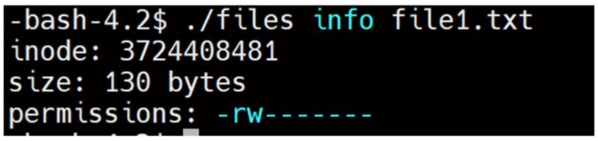
symlink: This action takes two filenames as arguments and creates a symbolic link from the second filename to the first filename using the “symlink()” system call.

rm: This action takes a filename as argument and removes the file using the “unlink()” system call. If the file is a directory, it prints an error message and does not remove the directory.

The code also provides proper error messages using the “perror()” function in case an action fails to execute. The usage message is also printed if the tool is invoked with the wrong syntax.

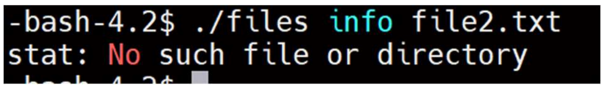
Result:

info(success):



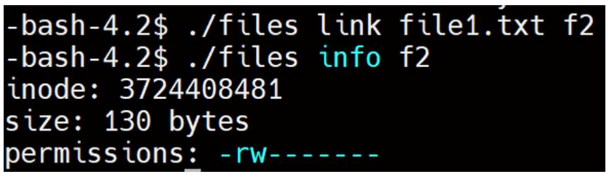
```
-bash-4.2$ ./files info file1.txt
inode: 3724408481
size: 130 bytes
permissions: -rw-------
```

info(failed):



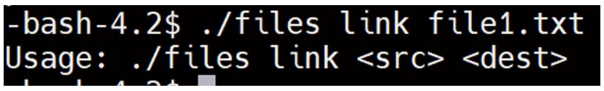
```
-bash-4.2$ ./files info file2.txt
stat: No such file or directory
```

link(success):



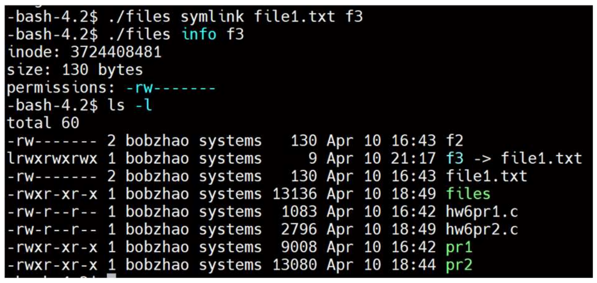
```
-bash-4.2$ ./files link file1.txt f2
-bash-4.2$ ./files info f2
inode: 3724408481
size: 130 bytes
permissions: -rw-------
```

link(failed):



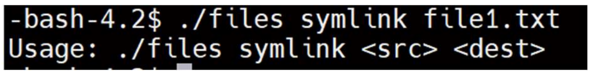
```
-bash-4.2$ ./files link file1.txt
Usage: ./files link <src> <dest>
```

symlink(success):



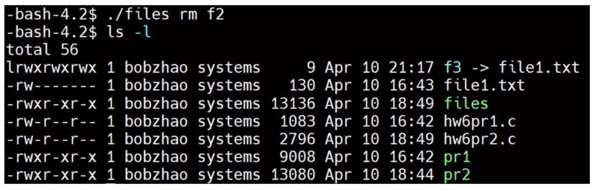
```
-bash-4.2$ ./files symlink file1.txt f3
-bash-4.2$ ./files info f3
inode: 3724408481
size: 130 bytes
permissions: -rw-------
-bash-4.2$ ls -l
total 60
-rw------- 2 bobzhao systems   130 Apr 10 16:43 f2
lrwxrwxrwx 1 bobzhao systems     9 Apr 10 21:17 f3 -> file1.txt
-rw------- 2 bobzhao systems   130 Apr 10 16:43 file1.txt
-rwxr-xr-x 1 bobzhao systems 13136 Apr 10 18:49 files
-rw-r--r-- 1 bobzhao systems  1083 Apr 10 16:42 hw6pr1.c
-rw-r--r-- 1 bobzhao systems  2796 Apr 10 18:49 hw6pr2.c
-rwxr-xr-x 1 bobzhao systems  9008 Apr 10 16:42 pr1
-rwxr-xr-x 1 bobzhao systems 13080 Apr 10 18:44 pr2
```

symlink(failed):



```
-bash-4.2$ ./files symlink file1.txt
Usage: ./files symlink <src> <dest>
```

rm(success):



```
-bash-4.2$ ./files rm f2
-bash-4.2$ ls -l
total 56
lrwxrwxrwx 1 bobzhao systems     9 Apr 10 21:17 f3 -> file1.txt
-rw------- 1 bobzhao systems   130 Apr 10 16:43 file1.txt
-rwxr-xr-x 1 bobzhao systems 13136 Apr 10 18:49 files
-rw-r--r-- 1 bobzhao systems  1083 Apr 10 16:42 hw6pr1.c
-rw-r--r-- 1 bobzhao systems  2796 Apr 10 18:49 hw6pr2.c
-rwxr-xr-x 1 bobzhao systems  9008 Apr 10 16:42 pr1
-rwxr-xr-x 1 bobzhao systems 13080 Apr 10 18:44 pr2
```