Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 10: System-Level I/O
May 23, 2024

Practice Problems

Exercise 10.1. What is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd);
    exit(0);
}
```

Solution: Assuming that the program was directly executed by the shell and inherited from it the stdin, stdout, and stderr file descriptors (0, 1, and 2, respectively), then `foo.txt` would be opened with file descriptor 3. Since it was subsequently closed, it is now available for the next file, which in this case is `baz.txt`. Thus, `fd2` has value `3`.

Exercise 10.2. Suppose the disk file `foobar.txt` consists of the six ASCII characters `foobar`. Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Solution: Because `open()` was called twice on the same file, both file descriptors point to the same underlying file in the v-node table, but they point to the different files on the open file table. Therefore, their file offsets (position) are different. The output will be: `f`.

Exercise 10.3. As before, suppose the disk file `foobar.txt` consists of the six ASCII characters `foobar`. Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd;
    char c;

    fd = Open("foobar.txt", O_RDONLY, 0);
    if (Fork() == 0) {
        Read(fd, &c, 1);
        exit(0);
    }
    Wait(NULL);
    Read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Solution: The process begins by opening `foobar.txt`. When it calls `Fork()`, the child process inherits its file descriptors, and they point to the same entries on the file table. The parent suspends by calling `Wait(NULL)`, allowing the child to read from the descriptor first, and thus moving the file position forward by 1 byte. When the parent has reaped the child, its call to `Read()` causes a byte at position 1 to be read, and that character is `o`.

Exercise 10.4. How would you use `dup2` to redirect standard input to descriptor 5?

Solution: The standard input stream's file descriptor is normally given by `STDIN_FILENO`, which is exposed in `unistd.h`. To redirect standard input to descriptor 5, we must close the v-node associated with the standard input stream, and have the `STDIN_FILENO` file descriptor point to same entry in the open file table as file descriptor 5, like so:

```
dup2(5, STDIN_FILENO);
```

Exercise 10.5. Assuming that the disk file `foobar.txt` consists of the six ASCII characters `foobar`, what is the output of the following program?

```
#include "csapp.h"

int main()
```

```
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd2, &c, 1);
    Dup2(fd2, fd1);
    Read(fd1, &c, 1);
    printf("c = %c\n", c);
}
```

Solution: Throughput the entire program, `fd1` and `fd2` always point to the same underlying file (i.e., v-node). However, their entries in the open file table are distinct before the call to `Dup2`, which means their file offsets are different.

When `Read()` is used on `fd2`, it moves its file pointer forward by 1, so that the next character to be read is `o`. When `Dup2(fd2, fd1)` is called, this causes `fd1` to be closed, and to be subsequently updated to point to the same entry as `fd2`. Therefore, it now shares the file offsets with `fd2`, making it so that the call to `Read()` with `fd1` results in reading `o`.

Exercise 10.6. What is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    fd2 = Open("bar.txt", O_RDONLY, 0);
    Close(fd2);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

Solution: Assuming that the program inherits the three standard file descriptors from the shell (0, 1, and 2), then `fd1` will be 3, and `fd2` will be 4. When `fd2` is closed, this makes file descriptor 4 again, which is thus reused when opening `baz.txt`. hence, the output is 4.

Exercise 10.7. Modify the `cpfile` program in Figure 10.5 so that it uses RIO functions to copy standard input to standard output, `MAXBUF` bytes at a time.

Exercise 10.8. Write a version of the `statcheck` program in Figure 10.10, called `fstatcheck`, that takes a descriptor number on the command line rather then a filename.

Solution: See `08-fstatcheck/fstatcheck.c`:

```c
#include <stdio.h> /* fprintf(), stderr */
#include <stdlib.h> /* exit(), EXIT_FAILURE */
#include <sys/stat.h> /* stat(), struct stat, S_ISREG(), S_ISDIR(), S_ISSOCK() */
#include <unistd.h>
#include <string.h> /* strerror() */
#include <errno.h> /* errno */

#define FD_ARG_IDX 1

void
unix_error(char *msg)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    /* Ensure we got enough arguments */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s file-descriptor\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Make sure it's a number */
    char *endp;
    errno = 0;
    int fd = (int) strtol(argv[FD_ARG_IDX], &endp, 0); /* any base */
    if (errno != 0)
        unix_error("Error reading input number");
    else if (*endp != '\0') {
        fprintf(stderr, "%s: Encountered non-numeric character %c in input string
            %s\n",
            argv[0], *endp, argv[FD_ARG_IDX]);
        exit(EXIT_FAILURE);
    }

    /* Read file metadata */
    struct stat sb;
    if (fstat(fd, &sb) == -1)
        unix_error("Failed to read metadata for input file descriptor");

    /* Determine file type */
    char *type, *readok;
    if (S_ISREG(sb.st_mode))
```

```
        type = "regular";
    else if (S_ISDIR(sb.st_mode))
        type = "directory";
    else
        type = "other";

    /* Check read access */
    if ((sb.st_mode & S_IRUSR))
        readok = "yes";
    else
        readok = "no";

    /* Display information */
    printf("type: %s, read: %s\n", type, readok);
    exit(EXIT_SUCCESS);
}
```

Exercise 10.9. Consider the following invocation of the `fstatcheck` program from Problem 10.8:

```
linux> fstatcheck 3 < foo.txt
```

You might expect that this invocation of `fstatcheck` would fetch and display metadata for file `foo.txt`. However, when we run it on our system, it fails with a "bad file descriptor." Given this behavior, fill in the pseudocode that the shell must be executing between the `fork` and `execve` calls:

```
if (Fork() == 0) { /* child */
    /* What code is the shell executing right here? */
    Execve("fstatcheck", argv, envp);
}
```

Solution: It's possible that the shell closes file descriptor 0, corresponding to standard input, and opens the given file. The effect is that file descriptor 0 is available, so after the call to `Open("foo.txt")`, it points to the underlying file. Thus, file descriptor 3 is never used. The shell probably does the following:

```
if (Fork() == 0) { /* child */
    close(STDIN_FILENO);
    int fd = Open("foo.txt", O_RDONLY, 0); /* On file descriptor 0 probably */
    Execve("fstatcheck", argv, envp);
}
```

The following is likely what we expected would happen:

```
if (Fork() == 0) { /* child */
    int fd = Open("foo.txt", O_RDONLY, 0); /* Opens on file descriptor 3 */
```

```
    dup(fd, STDIN_FILENO); /* Closes STDIN_FILENO, and point to same entry as fd
        */
    Execve("fstatcheck", argv, envp);
}
```