CSE 422S: Studio 15

Linux Pipes and FIFOs

Xingjian Xuanyuan

September 27, 2022

September 27, 2022

In this studio, we will:

1. *Create and pass data through pipes with `pipe()` system call.*
2. *Create and pass data through FIFOs with the `mkfifo()` function.*
3. *Implement a rudimentary active object pattern with processes and FIFOs.*
4. *Use the `getline()` function to read a line of data at a time from a file into a dynamically allocated buffer, and then send it over a FIFO.*

First, create a C program that:

a Calls `pipe()` to create a pipe with read and write file descriptors

b Calls `fork()` to create a child process

c After the fork, the child process should close its copy of the read file descriptor with `close()`, and the parent process should close its copy of the write file descriptor similarly

d After the fork, the child process should write several test messages to the pipe

e After the fork, the parent process should read several messages from the pipe and print them to the standard output stream (`stdout`)

```c
/* pipe.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_LINE_NUM 10
#define MAX_LINE_LEN 100

int main()
{
    char msg[MAX_LINE_LEN];
    int i, pipefds[2];
    pid_t p;
    FILE *file_stream;

    if (pipe(pipefds) == -1) {
```

```
        fprintf(stderr, "Error: pipe()\n");
        exit(EXIT_FAILURE);
    }

    p = fork();
    if (p < 0) {
        fprintf(stderr, "Error: fork()\n");
        exit(EXIT_FAILURE);
    } else if (p > 0) {      // Parent process
        close(pipefds[1]);   // Close writing end

        if ((file_stream = fdopen(pipefds[0], "r")) == NULL) {
            fprintf(stderr, "Error, fdopen(parent)\n");
        }

        i = 0;
        while (fgets(msg, MAX_LINE_LEN, file_stream) != NULL) {
            printf("%d. %s", i++, msg);
        }
        printf("Parent with pid %u received from child: %s", getpid(),
↪   msg);
        fclose(file_stream);
        wait(NULL);          // Wait for child to exit
    } else {                 // Child process
        close(pipefds[0]);   // Close reading end

        if ((file_stream = fdopen(pipefds[1], "w")) == NULL) {
            fprintf(stderr, "Error: fdopen(child)\n");
            exit(EXIT_FAILURE);
        }

        fprintf(file_stream, "Hello from child process with pid %u!\n",
↪   getpid());
        for (i = 0; i < MAX_LINE_NUM; i++) {
            fprintf(file_stream, "%s\n", "We are walking in the air...");
        }

        fclose(file_stream);
    }

    return EXIT_SUCCESS;
}
```

Build and run the above program on my Raspberry Pi, I obtained the following terminal output:

```
0. Hello from child process with pid 3258!
1. We are walking in the air...
2. We are walking in the air...
3. We are walking in the air...
4. We are walking in the air...
5. We are walking in the air...
6. We are walking in the air...
7. We are walking in the air...
8. We are walking in the air...
```

```
9. We are walking in the air...
10. We are walking in the air...
Parent with pid 3257 received from child: We are walking in the air...
```

Now, we are going to implement a program that provides a rudimentary active object. An active object is an executable context that performs services for other contexts upon request. In this case, our active object will be a process that listens to one end of a FIFO, does some trivial computation when data is received, and prints the result of that computation to standard output. Any number of other processes can open the FIFO's read side and submit data, and the order that requests are scheduled depends on a race for the FIFO (recall that writes and reads are atomic for less than `PIPE_BUF` sized data).

To begin, create a new file for the program that should:

a Create a new FIFO with the function `mkfifo()`, giving it both read and write permissions (`S_IRUSR | S_IWUSR`). This is documented at `man 3 mkfifo`. *Note: if the FIFO already exists, the call to `mkfifo()` will fail and `errno` will be set to `EEXIST`, in that case your program should simply continue to that next step.*

b Open the FIFO for reading (as though it were a file) using `fopen()`.

c Enter a `while` loop that continually attempts to read from the FIFO until EOF is read. Whenever something is read, print it to the standard output.

```c
/* fifo.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    char *my_file = "LochLomond";
        int c;
        FILE *file_stream = NULL;

        if (mkfifo(my_file, S_IRUSR | S_IWUSR) == -1) {
                fprintf(stderr, "Error: mkfifo %d\n", errno);
                exit(EXIT_FAILURE);
        }

        if ((file_stream = fopen(my_file, "r")) == NULL) {
                fprintf(stderr, "Error: fopen fifo\n");
                exit(EXIT_FAILURE);
        }

        while ((c = fgetc(file_stream)) != EOF) {
            fprintf(stdout, "%c", c);
        }

        fclose(file_stream);

        return EXIT_SUCCESS;
}
```

If we build and run the program, it will create a FIFO in the working directory:

```
$ file LochLomond
LochLomond: fifo (named pipe)
```

In a separate terminal, we can type in the command:

```
echo "But the broken heart it kens, nae second spring, Tho' the waefu' may
↪  cease frae their greetin'!" > LochLomond
```

which will insert the string following `echo` into the FIFO. Alternatively, we can insert a file using the command:

```
cat pipe.c > LochLomond
```

Now, we will make two modifications to your active object program. After opening a read stream to the FIFO, also open a write stream. Your active object will not ever write to this stream, but the fact that it is held open will prevent your program from automatically quitting.

Next, modify the active object to read from the FIFO with the function `fscanf()`. This allows you to perform formatted input. Your program should read integer inputs from the pipe, double the input, and then print out the original and the new values.

Finally, write two programs that open the write end of the FIFO with `fopen(fifo_name, ""w)` and insert integer values with `fprintf(())`. One of these programs should only insert even numbers, and the other should only insert odd numbers.

```c
/* fifo_rw.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
        char *my_file = "Integers";
        int c;
        FILE *file_stream_r = NULL, *file_stream_w = NULL;

        if (mkfifo(my_file, S_IRUSR | S_IWUSR) == -1) {
                fprintf(stderr, "Error: mkfifo %d\n", errno);
                exit(EXIT_FAILURE);
        }

        if ((file_stream_r = fopen(my_file, "r")) == NULL) {
                fprintf(stderr, "Error: fopen fifo for read\n");
                exit(EXIT_FAILURE);
        }
        if ((file_stream_w = fopen(my_file, "w")) == NULL) {
                fprintf(stderr, "Error: fopen fifo for write\n");
                exit(EXIT_FAILURE);
        }

        while (fscanf(file_stream_r, "%d", &c) == 1) {
                fprintf(stdout, "Original: %d; New: %d.\n", c, 2*c);
        }
```

```
        fclose(file_stream_r);
        fclose(file_stream_w);

        return EXIT_SUCCESS;
}
```

```
/* fifo_even.c */
#include <stdio.h>
#include <stdlib.h>

const char *fifo_name = "Integers";

int main()
{
        int d;
        FILE *even_int;
        even_int = fopen(fifo_name, "w");
        if (even_int == NULL) {
                fprintf(stderr, "Error: fopen fifo\n");
                exit(EXIT_FAILURE);
        }
        while (fscanf(stdin, "%d", &d) == 1) {
                if (d % 2 != 0) continue;
                if (fprintf(even_int, "Original: %d; New: %d.\n", d, 2*d) >
 ↪  0) {
                        printf("Successfully insert %d.\n", d);
                }
        }
        fclose(even_int);

        return EXIT_SUCCESS;
}
```

```
/* fifo_odd.c */
#include <stdio.h>
#include <stdlib.h>

const char *fifo_name = "Integers";

int main()
{
        int d;
        FILE *odd_int;
        odd_int = fopen(fifo_name, "w");
        if (odd_int == NULL) {
                fprintf(stderr, "Error: fopen fifo\n");
                exit(EXIT_FAILURE);
        }
        while (fscanf(stdin, "%d", &d) == 1) {
                if (d % 2 != 1) continue;
                if (fprintf(odd_int, "Original: %d; New: %d.\n", d, 2*d) >
 ↪  0) {
                        printf("Successfully insert %d.\n", d);
```

```
                }
        }
        fclose(odd_int);

        return EXIT_SUCCESS;
}
```

For the final task, create a new FIFO writer program that again shares the same FIFO with the active object reader program as in the previous exercise. Have your new FIFO writer program take the name of a standard file (i.e., one on disk) as a command line argument, open the file, and then repeatedly: use the `getline()` library function to move a lone of text out of the file into a dynamically allocated memory buffer, and then output the entire line into the FIFO, until the end of the file is reached. *When the end of the file is reached, your new FIFO writer program should free the dynamically allocated memory buffer.*

```
/* fifo_stdd.c */
#include <stdio.h>
#include <stdlib.h>

const char *fifo_file = "Integers";
unsigned int buf_len = 100;
char *buffer = NULL;

int main(int argc, char *argv[])
{
        int counter = 0;
        FILE *fp_fifo, *fp_stdd;
        fp_fifo = fopen(fifo_file, "w");
        fp_stdd = fopen(argv[1], "r");
        if (fp_fifo == NULL || fp_stdd == NULL) {
                fprintf(stderr, "Error: fopen\n");
                exit(EXIT_FAILURE);
        }

        buffer = (char *)malloc(buf_len * sizeof(char));
        while (getline(&buffer, &buf_len, fp_stdd) != -1) {
                if (fprintf(fp_fifo, "%s\n", buffer) > 0) {
                        printf("Successfully insert line %d of size %d.\n",
 ↪  ++counter, buf_len);
                }
                buffer = realloc(buffer, buf_len * sizeof(char));
        }

        free(buffer);

        fclose(fp_fifo);
        fclose(fp_stdd);

        return EXIT_SUCCESS;
}
```