Interprocess Communication

DEPARTMENT OF COMPUTER ENGINEERING

In a previous lab session we looked at how to create a new process using *fork()* function. In this lab session we'll look at how to establish communication between these processes. But first of all we'll look at some basic input output methods.

1. Lower level I/O

During this lab we will be using a very basic method of file I/O that uses the C functions *open()*, *read()*, *write()* and *close()*. Open is similar to fopen. These functions are the building blocks that other I/O operations are built on. For example, the file interface in C library consists of *fread()*, *fwrite()* and *fclose()* functions that are analogous *read()*, *write()* and *close()* system calls.

☞ *NOTE: If you have not used file interface in C library &/or not familiar with its usage, please use the online tutorials provided in the course page on FEeLS to learn about those functions.*

A basic example of the use of these functions is given below. This program is available as example1.1.c in the tarball given.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

int main() {
   char* banner = "This is a string written to a textfile by a C program\n";
   int desc = open("out.txt", O_WRONLY|O_APPEND|O_CREAT , S_IRUSR | S_IWUSR);
   write(desc,banner,strlen(banner));
   close(desc);
   return 0;
}
```

Exercise1.1:

a. Explain what the flags O_*WRONLY*, *O_APPEND* and *O_CREAT* do.
b. Explain what the modes *S_IRUSR*, *S_IWUSR* do.

A more advanced program is available as example1.2.c

The connection between the file interface (*fread()*, *fwrite()*, etc) and the lower level I/O is that, internally the file interface uses the lower level I/O to read and write, while providing more advanced functionality to make I/O easier and more efficient using buffering.

- The operating system alone handles most of the resources, including input output. The other programs just place the necessary arguments and execute a system call.
- Since the system calling convention of the operating system differs from the convention of C language, the libC library acts as an intermediate to translate between the two.
- The libC library functions for reading and writing a file are *read()* and *write()* respectively.

The return value of an *open()* function call is an integer called "file descriptor" (as opposed to the "file pointer" returned by an *fopen()* call). The file descriptor is a value given by the operating system to represent the open file. The same file descriptor number could refer to two different files at the same time in two different processes while the same process could have two file descriptors that represent the same file.

- Most of the resources in the computer are regarded as files in Unix like systems.
- Unix like systems has a simple and consistent interface for I/O where most of the work could be done by *read()* and *write()* function calls.

Exercise1.2:

a. Write a program called `mycat` which reads a text file and writes the output to the standard output.
b. Write a program called `mycopy` using *open()*, *read()*, *write()* and *close()* which takes two arguments, viz. source and target file names, and copy the content of the source file into the target file. If the target file exists, just overwrite the file.

2. Pipes

Pipes are a data channel that can be used for interprocess communication. Pipes could be of two types: *Named pipes* and *unnamed pipes*. Unnamed pipes could be used to communicate between a parent process and its' children or among siblings, whereas a named pipe could be used to communicate between any two processes. A named pipe is also called a FIFO. In this section we are looking at unnamed pipes.

Unnamed pipes

A pipe could be created with the function *pipe(int[2])*, which sets the two integers of the array into two file descriptors. The zeroth element of the input integer array becomes the read-end while the first element becomes the write-end.

Data written to the write-end of the pipe is buffered by the kernel until it is read from the read-end of the pipe. Here is a simple example of an unnamed pipe. This is also available as example2.1.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
	int pipe_ends[2];
	pid_t pid;
	int status;

	if(pipe(pipe_ends)){
		perror("Create pipe");
		return -1;
	}
	pid = fork();
	if(pid < 0){
		perror("Fork");
		return -1;
	}

	if(pid > 0){ /* parent */
		close(pipe_ends[0]);
		write(pipe_ends[1],"Hello!\n",8);
		close(pipe_ends[1]);
	}
	if(pid == 0){ /* child */
		char buff[128];
		close(pipe_ends[1]);
		int count = read(pipe_ends[0], buff, 128);
		buff[count] = '\0';
		printf("%s",buff);
	}

	wait(&status);
	return 0;
}
```

A more complex example is given as example2.2.c

Exercise2.1:
Look at the code in example2.2.c and answer the following questions.

a. What does `write(STDOUT_FILENO, &buff, count);` do?
b. Can you use a pipe for bidirectional communication? Why (not)?
c. Why cannot unnamed pipes be used to communicate between unrelated processes?
d. Now write a program where the parent reads a string from the user and send it to the child and the child capitalizes each letter and sends back the string to parent and parent displays it. You'll need two pipes to communicate both ways.

3. *dup()* and *dup2()* System Calls

Exercise3.1:
Write a program that uses `fork()` and `exec()` to create a process of `ls` and get the result of `ls` back to the parent process and print it from the parent using pipes. If you cannot do this, explain why.

When `exec()` is used to replace the execution image of a forked child, all the communication means are lost since `exec()` replaces all the original code. To solve this problem, `dup()` or `dup2()` function calls can be used.

`dup()` and `dup2()` are functions that can be used to redirect I/O. What it basically does is taking two file descriptors—say, descriptor1 and descriptor2—and making descriptor2 equivalent to descriptor1. i.e., writing to descriptor2 will result in writing to the file pointed by descriptor1.

Here's an example of using `dup2()` to redirect standard input and output into two files. This example is available as example3.1.c (you will also need the "fixtures" file provided in the tarball).

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
 * Executes the command "grep 'New Zealand' < fixtures > out".
 */

int main(int argc, char **argv)
{
        int in, out;
        char *grep_args[] = {"grep", "New Zealand", NULL};

        // open input and output files
        in = open("fixtures", O_RDONLY);
        out = open("out",
             O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);

        // replace standard input with input file
        dup2(in, 0);

        // replace standard output with output file
        dup2(out, 1);

        // close unused file descriptors
        close(in);
        close(out);

        // execute grep
        execvp("grep", grep_args);
}
```

A more complex example involving `dup()` is given as example3.2.c (pay attention to error checking as well).

Exercise3.2:

a. What does 1 in the line `dup2(out,1);` in the above program stands for?

b. The following questions are based on the example3.2.c
   i. Compare and contrast the usage of `dup()` and `dup2()`. Do you think both functions are necessary? If yes, identify use cases for each function. If not, explain why.
   ii. There's one glaring error in this code (if you find more than one, let me know!). Can you identify what that is (hint: look at the output)?
   iii. Modify the code to rectify the error you have identified above.

c. Write a program that executes "`cat fixtures | grep <search_term> | cut -b 1-9`" command. A skeleton code for this is provided as exercise3.2.c_skel.c. You can use this as your starting point, if necessary.

4. Named pipes

Named pipes are similar to unnamed pipes except a named pipe could be used to communicate between two unrelated processes. Creating a named pipe actually creates a file in your hard disk through which the processes could communicate. Here is a simple example. This example is available as example4.1reader.c and example4.1writer.c.

Example4.1reader.c:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_SIZE 1024

int main()
{
    int fd;
    char* fifo = "/tmp/fifo";
    char buf[MAX_SIZE];

    mkfifo(fifo,0666);

    fd = open(fifo, O_RDONLY);
    read(fd, buf, MAX_SIZE);
    printf("message read = %s\n", buf);
    close(fd);

    return 0;
}
```

Example4.1writer.c

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;
    char * fifo = "/tmp/fifo";

    mkfifo(fifo,0666);

    fd = open(fifo, O_WRONLY);
    write(fd, "Hi",2);
    close(fd);

    unlink(fifo);

    return 0;
}
```

Note that both `read()` call and `write()` calls on a named pipe (fifo) are blocking calls. Confirm this by invoking the two programs in alternating order (i.e., invoke the writer first, and then the reader; in the next iteration, invoke the reader first).

Exercise4.1:

a. Comment out the line "`mkfifo(fifo,0666);`" in the reader and recompile the program. Test the programs by alternating which program is invoked first. Now, reset the reader to the original, comment the same line in the writer and repeat the test. What did you observe? Why do you think this happens? Explain how such an omission (i.e., leaving out `mkfifo()` function call in this case) can make debugging a nightmare.

b. Write two programs: one, which takes a string from the user and sends it to the other process, and the other, which takes a string from the first program, capitalizes the letters and send it back to the first process. The first process should then print the line out. Use the built in command `tr()` to convert the string to uppercase.

NOTE: A named piped could be used with multiple readers &/or writers. Use the examples (`speak` and `tick`) given in FIFO page in Beej's Guide to IPC to test this functionality (e.g., Run multiple instances of readers and a single writer, and see which reader gets what's written. Then run a single instance of a reader and multiple instances of writers, and see what the reader gets).