COP4610 / CGS5765 Operating Systems Project 3
Process Synchronization

Important Notes

1. The project is based on the Linux operating system, not the Xv6.
2. Make sure your program works on `linprog.cs.fsu.edu` because that is where it will be graded.

Programming Task

In this assignment, you will learn how to use `fork` to create processes, and how to use Linux `pipe`s and shared memory for process synchronization. Specifically, you will create a program called `trans`. The program takes two command line parameters, an input file and an output file. It copies the input file to the output file using the shared memory. For example, to invoke the program, type

```
trans input-file output-file
```

at the command line prompt.

Here is how the program works: your program will create two processes, a parent process and a child process, using `fork`. These two processes share two pipes, one pipe carries data from the parent to the child and the other carries data in the opposite direction (a regular pipe is unidirectional.) The parent and the child also shares a 4KB memory. To transfer the file, the parent first reads a block of 4KB data from the input file into the shared memory (the last block of the data might be less than 4KB if the file size is not a multiple of 4KB), and sends the block number and the block length to the child through its pipe. After receiving the block number and the block length, the child writes the data from the shared memory to the output file, and the sends the block number back to the parent as an acknowledgment through the other pipe. After receiving the correct block number from the child, the parent continues to transfer the next block. This process continues until the whole input file has been written to the output file. The following are some specific requirements:

1. Put all your code in a single file, called `trans.c`. Note that programs using the POSIX shared memory API must be compiled with `cc -lrt` to link against the real-time library, `librt`. You can use the following command to compile your code:

   ```
   gcc -std=c99 -Wall -Wextra -D_XOPEN_SOURCE=700 -o trans trans.c -lrt
   ```

2. Your code must work with both binary and text files. For example, you can test it with the generated `trans` file, which is a binary file. You may use `fread`/`fwrite` to access files. It should also print out a helpful error message. It should also promote the user whether to overwrite the file if the output file already exists.

3. Make sure the input and output files are the same after running `trans`. You can compare their checksums using `shasum`.

4. Use Linux pipe to create pipes. You can find the API and an example program at `http://man7.org/linux/man-pages/man2/pipe.2.html`.

5. Use the POSIX shared memory, **NOT** SysV shared memory. That is, your program should NOT use SysV APIs such as `shmget` and `shmat`. Instead, use `shm_open` and `mmap`. POSIX shared memory is the preferred way to use shared memory on Linux.

   You can find the man pages for POSIX shared memory at `http://man7.org/linux/man-pages/man7/shm_overview.7.html`, and some example code at `http://www.cse.psu.edu/~deh25/cmpsc473/notes/OSC/Processes/shm.html` (use the last three files; the first two files are not complete.).

   The shared memory must be created after the process has forked.

   When you name your shared memory object, include your fsuid to avoid conflicts. For example, use `/fsuid_cop4610` as the name in `shm_open`.

6. The data is transferred in blocks. Each block is 4KB bytes (except maybe the last one). The block number counts how many blocks have been transferred. It starts from 1. So, if you are going to transfer a file of 10KB, the block numbers are 1,2, and 3.

   The block numbers and the block lengths should be sent and received as the standard C `int`. On `linprog`, it will be a 64-bit integer since `linprog` uses 64-bit linux. On a 32-bit Linux system, they will be 32-bit integers.

   After all the blocks have been transferred, the parent process sends block number 0 and length 0 to the child. After receiving block number 0, the child replies the parent with block number 0 and then exit. The parent receives the child's acknowledgment to block 0 and exits too.

   Therefore, the protocol to transfer the file is similar to the tftp protocol. For example, to copy a 6KB file, the parent writes the first 4KB data of the file to the shared memory and send two integers, 1 and 4096, to the child via the pipe. The child receives these two numbers, copies 4096 bytes from the shared memory to the output file, and sends back 1 to the parent via the other pipe. After receiving 1, the parent copies the left 2KB data to the shared memory and send 2 and 2048 to the child. The child receives them from the pipe, copies 2048 bytes to the output file, and replies with 2 to the parent. The parent then send 0, 0 to the child. The child receives 0 and replies with a 0 and then exit. The parent receives 0 and exits too. It is clear that in this protocol only one process can access the shared memory at a time.

7. Your code should include necessary error handling. For example, it should check the errors for `fork`, pipes, and shared memory. Both examples given previously have the necessary error handling.