CS 3113: Project 2

Objectives

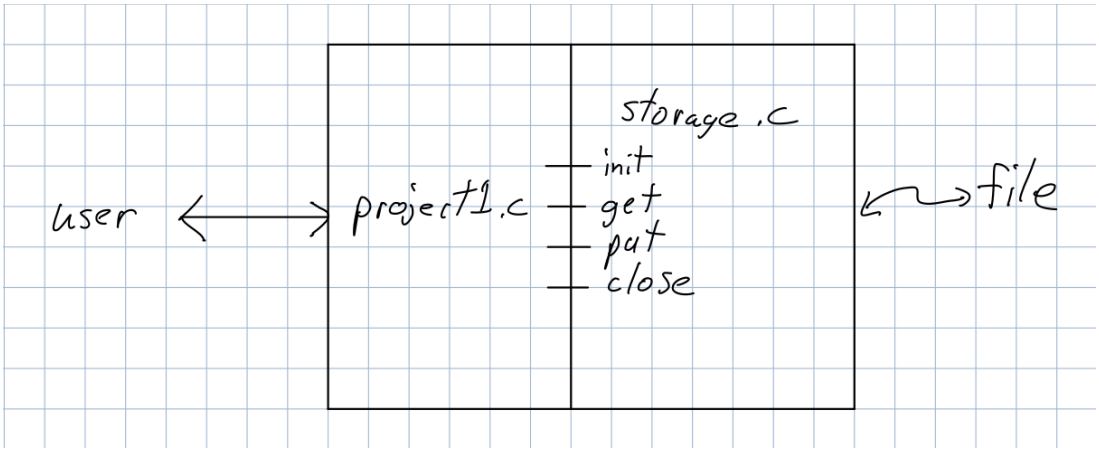
The objectives of this project are for you to:

- Gain experience in transferring data over unix pipes.
- Gain experience in implementing communication protocols.

Overview

For project 1, you implemented a program composed of two pieces:

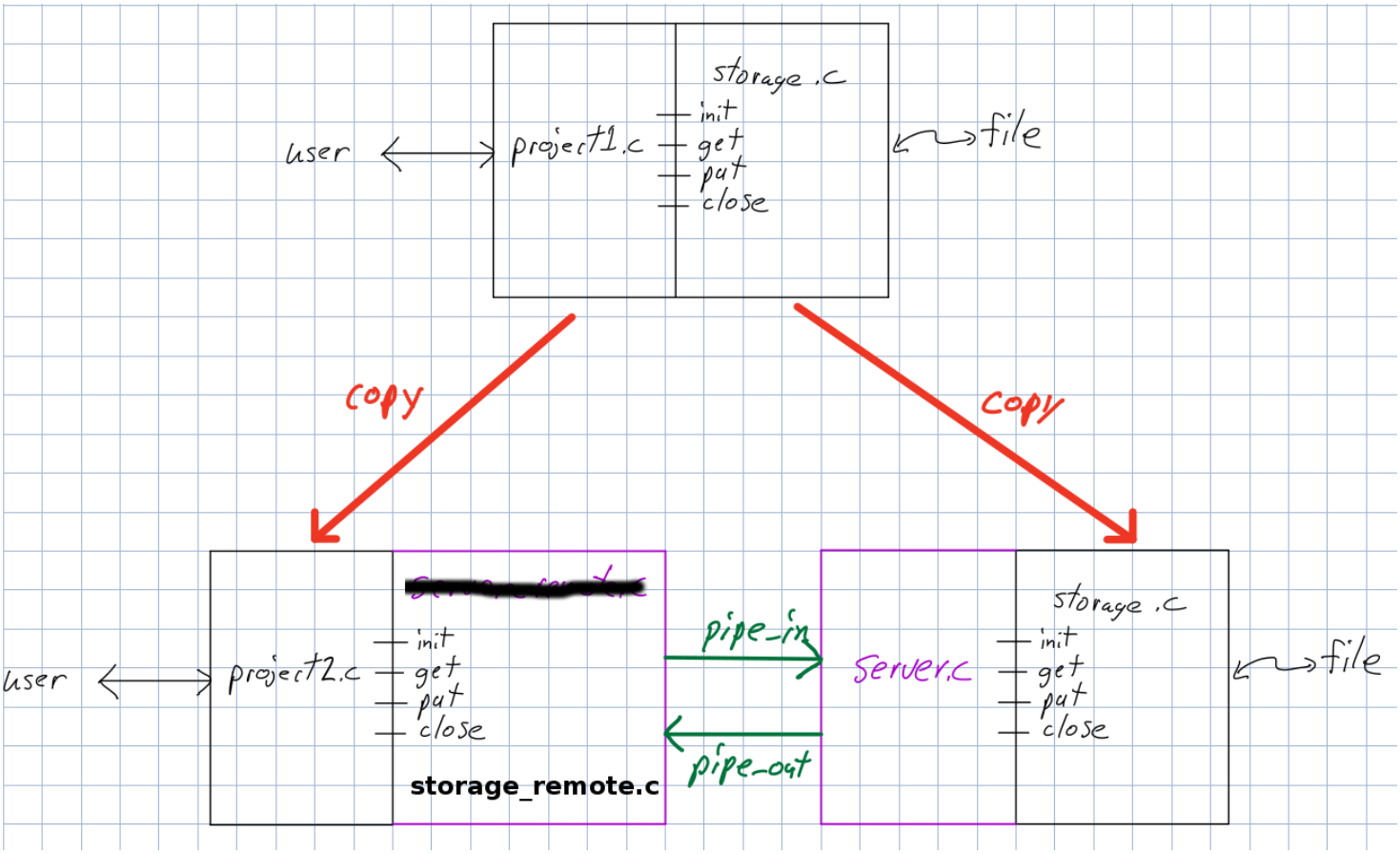
- A storage API that allows a program to read data from and write data to specific parts of a data file.
- A text-based user interface that enabled a user to manipulate the contents of an in-memory buffer and then to store this buffer to the file.



In particular, the API provided a specific set of functions for manipulating the storage file.

For project 2, we will keep these two pieces the same, except that they will become their own processes and the interaction between them will be through a pair of named pipes. Specifically, we will implement a client-server model of interaction, with the user interface acting as the client and the storage component acting as a server.

- project2.c will be a direct copy of project1.c, with three changes (documented below). It will be linked with server_remote.c to create the client process.
- storage.c/h will remain identical
- The server process will use storage.c/h and accept commands from the client over a named pipe and return data over a second named pipe.
- project2.c



Proper Academic Conduct

The code solution to this project **must be done individually.** Do not copy solutions to this problem from the network and do not look at or copy solutions from others. However, you may use the net for inspiration and you may discuss your general approach with others. These sources must be documented in your README file.

Specification

Named Pipes

Add a rule in your Makefile to create two named pipes in the local directory. In Linux, a named pipe can be created at the command line through this command:

```
mkfifo PIPE_NAME
```

where PIPE_NAME is the name of the pipe as it will appear in the file system.

Your two pipes are:

- **pipe_in**: used to send information **from** the client **to** the server
- **pipe_out**: used to send information **from** the server **to** the client

In C, you can open named pipes just as you would any file, including specifying whether you will open the pipe for reading or writing (note that a process should only choose one of these).

A word of caution: the open() system call on a pipe will block until the other end is also opened. For example, if your process opens a named pipe for writing, the open() system call will not return until after another process has opened the same pipe for reading. One implication is that if two processes are trying to open a pair of pipes (one to read and the other to write), make sure that you open them in the same order. **For this project, you must open pipe_in first and then pipe_out**.

Storage API

The storage API does not change from the prior project.

Storage_Remote API

The Storage_Remote API provides **exactly** the same functions (names, parameters and return values) as the Storage API. Once implemented, you will be able to link it with your project2.c code with no real changes to your code.

The key difference with the Storage API is that this new API interacts with the server process through the pipes in order to implement its four key functions. The protocol for interacting with the server is defined below.

- `storage_remote.h` is provided
- Implement `storage_remote.c`

---

project2

project2.c (along with the Storage_Remote API) implements the client process. This file is a copy of project1.c with the following changes:

- Include `storage_remote.h` in place of `storage.h`
- During the linking phase that creates the project2 executable, link to `storage_remote.o` instead of `storage.o`
- If no arguments are given to the `project2` executable, then it must specify the storage file `storage.bin`. If one argument is given, then this argument is used as the storage file name. If two or more arguments are given, then this program should indicate an error and exit.

Note that the **STORAGE** structure in `storage_remote.h` has changed slightly:

```
typedef struct
{
  int fd_to_storage;
  int fd_from_storage;
} STORAGE;
```

It now stores file descriptors for information going to the storage server and coming from the storage server.

---

server

`server.c` implements (along with `storage.c`) the server program. The server is an infinite loop that:

- Opens the named pipes
- Waits for an INIT_CONNECTION message
- Initializes the local storage
- Loops until SHUTDOWN is received:
  - Receives a message
  - Responds appropriately
    - READ_REQUEST: forwards the read request to the storage. Sends a DATA message back.
    - WRITE_REQUEST: forwards the write request to the storage. Sends an ACKNOWLEDGE message back.
    - SHUTDOWN: Sends an ACKNOWLEDGE message back. Sleeps for one second (sleep(1)), and then closes the storage and the named pipes.

Note that once you start the server, it will continue to execute, even after the client sends a SHUTDOWN message.

Messages

The key components of the communication protocol are defined in `comm.h`:

```
// Possible types of messages
typedef enum  {WRITE_REQUEST = 0, READ_REQUEST, ACKNOWLEDGE,
               DATA, SHUTDOWN, INIT_CONNECTION} MessageType;

// Message header
typedef struct
{
  MessageType type;   // Type of the message
  int len_message;    // Length of data sent in message after header
  int location;       // Location to read/write in file
  int len_buffer;     // Length of bytes to read/write in file
}HEADER;
```

Any message that is sent from one process to the other will be contain an instance of the HEADER type. The header contains the type of the message. If additional data needs to be sent in addition to the header, then the length of the data will be stored in HEADER.len_message. The HEADER will then be followed by exactly this number of bytes.

The header also contains HEADER.location and HEADER.len_buffer. These values are used for specifying in the remote file the offset and the length of the data to be copied into or out of the file.

You can create and initialize an instance of the header on the stack with the following code:

```
HEADER h;
h.type = SHUTDOWN;
h.len_message = 0;
```

```
h.location = -1;       // These value is not used for SHUTDOWN messages,
h.len_buffer = -1;     // but it is good practice to set them to known values
```

Communication Protocol

There are four different types of interactions that can occur between the client and the server, corresponding to the four API function implemented in `storage_remote.c`. All interactions are initiated by the client and always have a response from the server.

init_storage()

```
STORAGE * init_storage(char * name);
```

- Opens the two named pipes.
- Sends an INIT_CONNECTION message to the server. This message includes the name of the file to be opened by the server (make sure to send the null terminator at the end of the name).
- The server opens the specified storage file.
- The server responds with an ACKNOWLEDGE message. Note: if the server experiences a fatal error, it is fine for it to terminate immediately.
- Return the newly created STORAGE object.
- If there is an error, this function must return NULL.

close_storage()

```
int close_storage(STORAGE *storage);
```

- Sends a SHUTDOWN message to the server.
- The server responds with an ACKNOWLEDGE message.
- Close the two file descriptors for the pipes.
- Free the dynamically allocated structure.
- Returns 0 on success and -1 on error.

get_bytes()

```
int get_bytes(STORAGE *storage, unsigned char *buf, int location, int len);
```

- Sends the READ_REQUEST message to the server.
- The server queries the file and responds with a DATA message (including the requested data; note that only the requested length of data should be sent).
- Returns the number of bytes read, zero if at end-of-file.

put_bytes()

```
int put_bytes(STORAGE *storage, unsigned char *buf, int location, int len);
```

- Sends the WRITE_REQUEST message to the server (including the data to be written; note that only the data to be written may be sent).
- The server writes the data to the file at the specified location and responds with an ACKNOWLEDGE message.
- Returns the number of bytes written.

Notes

- Unless otherwise specified, your processes may terminate if an error occurs (we do not intend to test these cases).
- You may assume that the user will provide arguments in the correct numerical range.
- All program input is from STDIN.
- All normal program output is to be placed in STDOUT.
- You must address the case of incorrect numbers of arguments or of parse errors by writing an error to STDERR.

Testing

Your testing procedue will be as follows:

- Start the server process (**server** executable).
- In a separate terminal, start the client process (**project2** executable).
- The user will interact with the project through the project2 process. By the nature of our design, the text-based user interface is identical to that of project 1: it must respond to all the same commands in the same ways as the prior project. **In particular, the same "diff tests" can be used to test project 2.**

Notes:

- We will not test STDOUT of the server, so it is okay to have it print high-level debugging information.

- For the autograder, we will use the same "diff test" approach as was used in Project 0. We will use many of the same tests that we used in project 1, but most will involve reading/writing to the storage file.

Examples

The user interaction is identical to that in project 1, so the project 1 test cases still stand.

Submitting Your Program

- Your submission is composed of the files required to compile your client and server, named exactly as shown (including casing):
  - **project2.c**: your source file
    - Include your name and project number at the top of the file
  - **server.c**: your source file
    - Include your name and project number at the top of the file
  - **storage.c/h**: your library file.
    - Include your name and project number at the top of the C file
  - **storage_remote.c/h**: your library file.
    - Include your name and project number at the top of the C file
  - Include copies of **comm.h** and **storage_common.h**
    - Do not change these provided files
  - **Makefile**: does the following:
    1. **all**: compiles your two programs (this is the default behavior). The executables are called **project2** and **server**
    2. **clean**: deletes the executable files (project2 and server), any intermediate files (.o, specifically), and pipe_in and pipe_out
    3. **pipes**: create the pipe_in and pipe_out named pipes
  - **README.txt**: documentation
    - Include your name and project number at the top of the file
    - Document any Internet resources that you consulted (URLs)
    - Document any peer class members that you discussed your solution with. Note that you may not look at or share code that solves this specific problem
- Documentation requirements: function-level and inline documentation are important components of writing code. They help you to organize your thoughts while programming and help to communicate the method and the intent of the code to your future self or to collaborators (of course, you will not have collaborators in this class). While we will not be specifically grading documentation in this class, **we will not be able to comment on your code unless it is sufficiently documented.** This will be true whether you are asking for help before the submission deadline or looking for feedback on your solution after the deadline. In short, you should take the time to properly document your code as you develop it.
- Submit your files:
  - Create a zip file named **project2.zip** containing your source code (c and h files), Makefile and README.txt file. The following Linux shell command will do this for you:

    ```
    zip project2.zip *.c *.h Makefile README.txt
    ```

    Note I: this is a good thing to add to your Makefile.
    Note II: do not include object files, executables or temporary files in your zip file.

    Note III: if you are using any other mechanism for zipping, make sure that you do not include the directory that contains all of the files (your zip file should only contain the listed files).
  - Submit your zip file to Gradescope (access via Canvas). Shortly after your submission, you will receive automated feedback and a grade for the *correctness* component of your grade.
  - You may submit multiple times until the deadline.
- To be counted as on time, your solution must be submitted to the Gradescope server by 11:45:00 pm on Thursday, Oct. 17th. Grade penalties will be imposed for late submissions (see the syllabus for the details).

Grading Criteria

- README.txt (10%): contains the required information
- Makefile (10%): satisfies the make file requirements
- Correctness (80%): passes all of the (hidden) unit tests

Note that 90/100 points will be graded automatically by Gradescope and the remaining 10/100 points are graded by a human)

- If you use non-constant global variables, then 10% will be deducted from you grade.
- **Bonus:** 10 additional bonus points are possible for this project. We will use identical tests for the full project and for the bonus. In order to receive the full bonus points, then you must achieve at least 40/90 points assessed by the automatic grader (in the testing, you will not see these 90 points, you will only see a report of 0 .. 10 bonus points). In order to receive the bonus, then you must turn in your project no later than Thursday, Oct. 3rd at 11:45:00 pm.

Downloads

The following file contains several header and skeleton C files: project2_skel.tar. These skeleton files are good starting points for the source files that you must implement.

We will make our project1 implementation available as a starting point, but only after the project 1 late deadline.

Hints

- Don't implement everything at once. Instead, get INIT_CONNECTION and SHUTDOWN working first.
- Any debugging code should only print to STDERR. This way, it won't interfere with our testing procedures.
- We will be testing your server with our client. So, do not deviate from the defined protocol.

Addenda

- 2019-09-25: updated the Testing section to talk about how you should be testing your program.
- 2019-09-28: use of (non-constant) global variables will result in a 10% point loss.

  Here is our our implementation of project 1.
- 2019-10-01: Most of the tests we are using on gradescope involve reading and/or writing to the storage file.

---

andrewhfagg at gmail.com

Last modified: Tue Oct 1 09:42:10 2019