CMPUT 379 - Assignment #2 (12%)

A Chat Program Using FIFOs (first draft)

Due (Phase 1): Monday, October 16, 2017, 09:00 PM
Due (Phase 2): Monday, October 23, 2017, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in developing client-server programs that utilize FIFOs for communication, and file locking for controlling access to the available FIFOs.

Part 1: Experimenting with FIFOs in a Unix Environment

The client-server program specified in Part 2 relies on FIFOs (named pipes) for interprocess communication. In the classroom, we talked about creating FIFOs using the `mkfifo` shell command, and discussed some of their properties. This part is intended to give you more experience with their behaviour. To answer each question, you may need to design, implement and run a suitable experiment on the lab machines. Below, processes A and B and the FIFOs are assumed to be owned by the same user. In addition, the processes have access to the FIFOs. So, unless a process chooses to open a FIFO in a restricted mode (e.g., a read-only mode), the process can read and write to the FIFO.

1. Assume that process A sends a long stream of data to process B through a FIFO. Can one detect the transmission activity by monitoring the "size" of the FIFO on the file system (e.g., by repeatedly issuing a `"ls -l"` command)?

2. Can two processes running on two different hosts in the lab communicate using a FIFO? As mentioned above, both processes and the FIFO are assumed to be owned by the same user, and both processes can open the FIFO.

3. Assume that A and B run on the same host. Process A first opens a FIFO in the `O_RDWR` mode and waits for input from the user (without writing to the FIFO). Subsequently, while A is waiting for the user input, process B attempts to open the FIFO in a `O_RDONLY` mode. Does B block when calling `open()`?

4. Assume that A and B run on the same host. Process A first opens a FIFO in the `O_RDWR` mode, puts a lock on the FIFO using `lockf()`, described in the UNIX manual pages, and then waits for input from the user. Can process B detect that the FIFO is locked while A is waiting for the user input?

5. Each of the following loops are intended to read one line from the `stdin`, and then echoes the line to the `stdout` using low level unbuffered I/O functions (commonly used in client-server programs). Which loop (if any) works properly? Explain.

(a)
```
int  len; char buf[80];
while(1) {
    len= read(STDIN_FILENO, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, strlen(buf));
    if (strstr(buf, "exit") != NULL) exit(0);
}
```

(b)
```
int  len; char buf[80];
while(1) {
    memset(buf, 0, sizeof(buf));
    len= read(STDIN_FILENO, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, strlen(buf));
    if (strstr(buf, "exit") != NULL) exit(0);
}
```

In the deliverables, you need to describe your experimental work, and explain your answers (**don't** submit any separate code files for this part).

Part 2: A Chat Program

In this part you are asked to write a C/C++ program, called `a2chat`, that implements the chat server and client functionalities specified below. The program can be invoked as a server using `"a2chat -s baseName nclient"`, or as a client using `"a2chat -c baseName"`.

Data transmission from clients to the server uses at most `NMAX` $(= 5)$ FIFOs called the **inFIFOs**: `baseName-1.in, ..., baseName-5.in`. Data transmission in the other direction (from the server to the clients) uses at most `NAMX` $(= 5)$ FIFOs called the **outFIFOs**: `baseName-1.out, ..., baseName-5.out`. For simplicity, you may create the needed FIFOs (using some selected value for the `baseName` argument) before running the programs.

The argument `nclient` specifies the maximum number of clients that can be connected to the server at any time (nclient $\leq$ NMAX=5).

Locking FIFOs

To guarantee that no two client programs write to the same inFIFO when sending data to the server, each client is required to find an unlocked FIFO to use for sending data to the server. If an unlocked inFIFO is found, say `baseName-3.in`, then the client uses the `lockf()` function (see the man pages) to put a POSIX lock on the inFIFO. Subsequently, the client uses the corresponding outFIFO (i.e., FIFO `baseName-3.out`) for receiving data from the server. Clients should unlock the held inFIFO when a connection is closed.

The Client Loop

On start, the client prompts the user to enter a command line by displaying a suitable prompt message (e.g., `"a2chat_client:"`). Each iteration of the client loop polls the `stdin` for a command line. The client sends each valid command line to the server, waits for the server's response line, and processes the response line. In addition, if the client is in a connected state, the client also polls the specific outFIFO used for carrying incoming data from the server. If connected,

the client also processes the chat lines received from the server. The prompt message should appear after displaying any data to the user.

The Server Loop

On start, the server displays a suitable message indicating the `nclient` limit specified on the command line.

```
Example: % a2chat -s fifo 3
         Chat server begins [nclient= 3]
```

Each iteration of the server loop polls the `stdin` and the inFIFOs for command lines. The server processes each command line, forms a response line (if needed), and sends the response line to the client. Commands that cause the server to forward a chat line to multiple recipients generate a line for each recipient. Note that all communication between clients **go through the server**; there is no direct communication between any two clients. The *exit* command issued from the server's `stdin` terminates the server.

On a successful execution of a client's command by the server, the server sends the client a response line that starts with `"[server]"` followed by some suitable information about the execution of the command (or, just `"[server] done"` if there is no such information). On a failed execution of the command, the server's response line starts with `"[server] Error:"` followed by an explanation of the error.

The Chat Protocol

A chat client utilizes the chat service by issuing a sequence of command lines from the following list. As mentioned above, each command line is typically transmitted to the server, and the client program waits to display the server's response.

1. **open username**: Request the server to open a new chatting session using the specified `username` as the user's name.

   ```
   Example: % a2chat -c fifo
            Chat client begins
            a2chat_client: open ibm
            FIFO [fifo-1.in] has been successfully locked by PID [15360]
            [server] User 'ibm' connected on FFIO 1
            a2chat_client:
   ```

   The command fails to open a new session in the following cases:

   (a) No unlocked inFIFO is available for the client to use. In this case, the client should **not** terminate, rather it should advise the user to try later, and reissues the prompt.

   (b) The client has a session already going on.

   (c) The server has reached the client limit specified by `nclient`.

   (d) There is another client that uses the specified `username` for chatting.

2. **who**: Get a list of the logged in users.

```
Example:  a2chat_client: who
          [server]: Current users: [1] ibm, [2] dell
```

   In the example, the response line indicates the FIFO number used in each session.

3. **to user1 user2** $\cdots$: Add the specified users to the list of recipients.

```
Example: a2chat_client: to apple ibm dell unknown
         [server] recipients added: apple, ibm, dell
```

   In the above example, there is no client with the name `"unknown"`. The server just ignores such username.

4. $<$ **chat_line**: Send chat_line to all specified recipients.

```
Example: a2chat_client: < Hello apple ibm dell
         a2chat_client:
         [dell] Hello apple ibm dell
         a2chat_client:
```

   In the example, user `dell` is both a sender and recipient. In response, the server forwards the chat line prefixed with the sender's name in brackets to all recipients.

5. **close**: Close the current chat session **without** terminating the client program. The user may subsequently open a new session with a different username.

```
Example: a2chat_client: close
         [server] done
         a2chat_client:
```

   Note: the server is **not** required to notify other clients that a session has been closed.

6. **exit**: Close the current chat session, and terminate the client program. If the `exit` command is received from a client, the server does not generate a response line. Otherwise (if the command is generated from the server's `stdin`), the server terminates.

More Details

1. This is an individual assignment. Do not work in groups.

2. Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In such cases, you should make reasonable design decisions that do not contradict what we have said and do not significantly change the purpose of the assignment. Document such design decisions in your source code, and discuss them in your report. Of course, you may ask questions about this assignment (e.g., in the Discussion Forum) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

3. When developing and testing your program, **make sure you clean up all processes before you logout of a workstation.** Marks will be deducted for processes left on workstations.

Deliverables

1. All programs should compile and run on the lab machines (e.g., ug[00 to 34].cs.ualberta.ca) using only standard libraries (e.g., standard I/O library, math, and pthread libraries are allowed).

2. Make sure your programs compile and run in a fresh directory.

3. Your work (including a Makefile) should be combined into a single tar archive 'submit.tar'.

   (a) Executing 'make' should produce the `a2chat` executable file.

   (b) Executing 'make clean' should remove unneeded files produced in compilation.

   (c) Executing 'make tar' should produce the 'submit.tar' archive.

   (d) Your code should include suitable internal documentation of the key functions.

4. **Phase 1 Deliverables:**

   (a) Typeset a text, HTML, or PDF file `"a2answers"` containing the answers to the questions in Part 1. Explain your answers. You **don't** need to submit any separate code files for this part.

   (b) Submit a restricted version of the chat program, called `a2rchat` with the following restrictions:

      i. For the **"open username"** command, the server is **not** required to check that another client is currently using the specified `username` for chatting, or that the client has a session already going on.

      ii. The restricted version is not required to support the **"to"** or **"who"** commands. The server just echoes each received chat line to the sender.

   Thus, the server in `a2rchat` works essentially as an **echo server** for each ongoing session.

   (c) There is no need to submit a project report. Describe the status of the project in a `"readme"` file.

5. **Phase 2 Deliverables**

   (a) Submit the complete version of the `a2chat` program.

   (b) Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:

   – **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)

   – **Design Overview:** highlight in point-form the important features of your design

   – **Project Status:** describe the status of your project; mention difficulties encountered in the implementation

   – **Testing and Results:** comment on how you tested your implementation

  - **Acknowledgments:** acknowledge sources of assistance

6. Upload your tar archive using the **Assignment #2 (phase 1 or 2) submission/feedback** links on the course's web page. For each phase, late submission is available for 24 hours for a penalty of 10% of the points assigned to the phase.

7. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

Phase 1 Deliverables:

20% : Answers to Part 1 questions

20% : Correct execution of the restricted chat program **a2rchat**

Phase 2 Deliverables:

10% : successful compilation of a complete program that is: modular, logically organized, easy to read and understand, and includes error checking after important function calls

03% : ease of managing the project using the makefile

40% : correctness of executing the **a2chat** program

07% : quality of the information provided in the project report

---