CMPUT 379 - Assignment #2 (7%)

FIFO-based Client-Server Programming
(An incomplete draft for the lab on February 8th)

Due: Wednesday, February 28, 2024, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience with developing system programs that utilize some of the following UNIX features: signals, threads, timers, pipes, FIFOs, and I/O multiplexing for nonblocking I/O.

Part 1

■ **Program Specifications.** This part asks for developing a C/C++ program that can be invoked as `% a2p1 nLine inputFile delay`. When run, the program performs a number of iterations. In each iteration it reads from the `inputFile` the next `nLine` text lines and displays them on the screen. It then immediately enters a delay period for the specified number of `delay` milliseconds. During the delay period, the program stays responsive to input from the keyboard, as detailed below. After the delay period ends, the program proceeds to the next iteration where it reads and displays the next group of `nLine` text lines, and then enters another delay period. Subsequent iterations follow the same pattern. More details are given below.

1. At the beginning (and end) of each delay period, the program displays a message to inform the user of its entering (respectively, leaving) a delay period.
2. During a delay period, the program loops on prompting the user to enter a command, and executes the entered command. So, depending on the length of the delay period, the user can enter multiple commands for the program to execute.
3. Each user command is either `quit`, or some other string that the program tries to execute as a shell command line. The `quit` command causes the program to terminate and exit to the shell. Other strings are processed by passing them to the shell using the standard I/O library function `popen` (See, e.g., Section 15.3 of the [APUE 3/E]).
4. Other than forking a child process implied by calling `popen()`, the program should not fork a child process to achieve the desired behaviour. The process, however, may have threads and/or use timers. Any timer used must work at millisecond-level granularity.

■ **Examples.** Example output will be posted on eClass. **Your output should follow closely the output of the example.**

■ **Deliverables.** Files `a2p1-AI-page1.png` and `a2p1-AI-page2.png` (posted online) contains code generated by some AI program to handle the above requirements (with no attempt for the output to look like the above example).

1. Answer the following questions in your report.
   (a) What is the purpose of using `SA_RESTART` in the code?

(b) Does the code satisfy the above requirements? Ignore the differences between the code's output and the given example. If no, identify at least two unsatisfied requirements.

2. Submit a working C/C++ `a2p1` program

Part 2

■ **Program Specifications.** You are asked to write a C/C++ program, called `a2p2`, that can be invoked either as a server using `% a2p2 -s`, or as a client using `% a2p2 -c inputFile`. Each entity (server or client) has a unique identification number (`idNumber`). The server's `idNumber` $= 0$, and the client's `idNumber` $= 1$. `inputFile` contains work that needs to be done by the client. In this part only one client communicates with the server.

Data transfers between the client and the server use FIFOs. A FIFO named `fifo-0-1` carries data from the server to the client. Similarly, `fifo-1-0` carries data in the other direction.

> **Note:** For simplicity, FIFOs may be created in the work directory using the shell command `mkfifo` prior to starting your program development.

Context. The client-server system in this part (continued in Part 3) is intended to model a simple *file sharing* (peer-to-peer) system that allows a community of users to share files contributed by members. In this model system, each user (a client program) uploads files of different objects (e.g., HTML files, images, sound clips, video clips, etc.) to be stored and distributed with the help of a server program. Each object has a unique name and an owner (the client program that succeeds in storing the file at the server).

Input File Format

The input file has the following format.

- A line starting with `'#'` is a comment line (skipped). Empty lines are skipped. For simplicity, an empty line has a single `'\n'` character.
- Else, a line that starts with a single digit (`idNumber` $= 1$) has format and meaning that matches one of the following cases:

  > **Note:** In this part, we have only one client having `idNumber` $= 1$, but the description below is more general and applies to both this part and Part 3.

  - `"idNumber (put|get|delete) objectName"`: only the client with the specified `idNumber` sends to the server the specified *get*, *put*, or *delete* request of the named object. An object name has at most `MAXWORD` $= 32$ characters.
  - `"idNumber gtime"`: only the client with the specified `idNumber` sends to the server a *get time* request.
  - `"idNumber delay x"`: only the client with the specified `idNumber` delays reading and processing subsequent lines of the input file for an interval of x milliseconds.

  - `"idNumber quit"`: only the client with the specified `idNumber` should terminate normally.

Each `"idNumber put objectName"` command line is followed by a block that specifies the object contents. Each such a block begins (ends) with a line that starts with the `'{'` character (respectively, the `'}'` character); no other information appears on the line. Inside a block, at most 3 content lines (each has at most 80 characters) may be given.

```
# A fragment of a transactions file
...
# file1 is empty
1 put file1
{
}

# file2 contains 3 text lines
1 put file2
{
  file2: line 1
  file2: line 2
  file2: line 3
}
...
```

The server keeps an `object` table that can store up to `NOBJECT= 16` objects (the `inputFile` has at most this number of objects). Each object has a name and its content lines. Initially, the object table is empty. Subsequently, as the client executes `put` (or `delete`) commands the server updates the table by adding (respectively, deleting) objects form the table. The server replies to a `"get objectName"` request by sending the stored information to the client (if the object exists in the table).

Packet Types

Communication in the system uses messages stored in formatted packets. Each packet has a type, and carries a (possibly empty) message. The program should support the following packet types.

- `PUT`, `GET`, and `DELETE`: For a specified object name, a client executes a get, put, or delete command by sending a packet of the corresponding type, where the message specifies the object name. An error condition arises at the server when the client's request asks for doing one of the following:
  - getting a non-existing object,
  - putting an object that already exists, or
  - deleting an object owned by another client (this case arises only in Part 3 where multiple clients may exist).
- `GTIME` and `TIME`: A client processes a *get server's time* command (`gtime`) by sending a `GTIME` packet (with an empty message). The server replies by sending a `TIME` packet where the message contains the time in seconds (a real number) since the server started operation.
- `OK` and `ERROR`: The server replies with an OK packet if the received request is processed successfully. Else, the server replies with an ERROR packet with a suitable error description in the message part.

The Client Loop

The client performs a number of iterations. In each iteration, it reads the next text line from the input file, and executes the specified command only if the client's `idNumber` matches the one specified on the line. Otherwise, the client ignores the line. The execution of a command depends on its type, as follows:

- Commands in the set {`put, get, delete, gtime`} are executed by sending a packet to the server to do the corresponding operation. The packet contains an object name in the message part, and the object's content lines (for a `put` command), as applicable. The client then waits for a server's response.
- The `delay` command is for the client to suspend its operation for the specified number of milliseconds (i.e., suspending reading and processing of subsequent input lines, and transmitting packets).

  > **Note:** For simplicity, (and unlike Part 1) here the client is **not** required to stay responsive to input from the keyboard, or do any other activity, during a delay interval.

- The `quit` command causes the client to terminate normally (and exit to the shell).

To monitor progress of the client program, the program is required to print information on

- All transmitted and received packets
- Received object content (in reply to a `GET` packet)
- when a client enters (and exits) a delay period

The Server Loop

The server loops on receiving and replying to the client's incoming requests. To monitor progress, the server is required to print information on

- All transmitted and received packets
- Received object content (in a `PUT` packet)

■ **Examples.** Example output will be posted on eClass.

■ **Deliverables.** Submit a working C/C++ `a2p2` program

Part 3

To be updated

More Details (all parts)

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

2. Only standard include files and libraries provided when you compile the program using `gcc` or `g++` should be used.

3. 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.

4. 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.

5. **Excused Absences (EAs) for this assignment.** Work on this assignment is allocated at least 15 days, and the assignment has three parts. I expect that you start as early as possible on the assignment, and complete and upload a solution to a new part every 5 days (on the average). You can modify your submitted solution archive many times before the due date.

   If you must miss part (or all) of the assignment then contact me as soon as possible. I typically ask for suitable documentation to substantiate an EA request (***EA is a privilege not a right***). If granted (and depending on the circumstances), I'll decide on using one of the following two ways.

   - I assess the percentage of the approved EA part, and calculate your mark based on your mark in the remaining parts. E.g., if the assignment carries 100 points, the excused part carries 25 points, you got 75α points in the remaining parts (for some $\alpha \in (0, 1]$), then you receive additional 25α points for the missed part.
   - I apply the general EA policy described in the course outline.

Deliverables (all parts)

1. All programs should compile and run on the lab machines (e.g., ug[00 to 34].cs.ualberta.ca) using only standard libraries.

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

3. Your work (including a Makefile and test files) should be combined into a single tar archive **'your_last_name-a2.tar'** or **'your_last_name-a2.tar.gz'**.

   (a) Executing 'make a2p1', 'make a2p2', or 'make a2p3' should produce the corresponding executable.

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

   (c) Executing 'make tar' should produce the above '.tar' or '.tar.gz' archive.

(d) Your code should include suitable internal documentation of the key functions. If you use code from the textbooks, or code posted on eclass, acknowledge the use of the code in the internal documentation. Make sure to place such acknowledgments in close proximity of the code used.

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

- Answers to the questions posted in each part
- **Acknowledgments:** (**Important!**) This section is a mandatory part in all submitted reports. If there is no particular source to acknowledge then state this in the section. Else, you need to transparently and honestly acknowledge all sources of assistance (including use of AI tools). Failure to do so is considered an act of cheating.

4. Upload your tar archive using the **Assignment #2 submission/feedback** link on the course's web page. Late submission (through the above link) is available for 24 hours for a penalty of 10%.

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

6. **Important:** You can check the integrity and completeness of your submission after uploading to eClass by downloading the submission, extracting the stored files in a fresh directory, and checking the extracted files.