CMPUT 379 - Assignment #2 (10%)

Concurrency and Client-Server Programming

Due: Thursday, March 2, 2023, 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 exhibits some concurrency aspects. The required program 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 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.

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

Part 2

File sharing software systems allow a community of users to share files contributed by members. This part calls for implementing a small set of features that may be used in a client-server system that provides such a service. In the 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).

■ **Program Specifications.** You are asked to write a C/C++ program that implements the functionality of a client-server system. The program can be invoked to run as a server using

```
% a2p2 -s
```

or, run as a client using

```
% a2p2 -c idNumber inputFile
```

where `idNumber` is a client identification number, and `inputFile` is a file that contains work that needs to be done by the clients. The server can serve at most `NCLIENT` $= 3$ clients. Each client is assigned a unique `idNumber` in the range $[1, \texttt{NCLIENT}]$. The server is assumed to have `idNumber` $= 0$.

Data transfers between each client and the server use FIFOs. Clients do not communicate with each other. A FIFO named `fifo-0-x` carries data from the server to client x. Similarly, `fifo-x-0` carries data in the other direction (from client to server).

> **Note:** For simplicity, the needed $2 \cdot$`NCLIENT` FIFOs may be created in the work directory using the shell command `mkfifo` prior to starting your program development.

Input File Format

The input file is a common file shared among all intended clients. The 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 specifies a command line whose format and meaning matches one of the following cases:
  - `"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.

Packet Types

Communication in the system uses messages stored in formatted packets. Each packet has a type, and carries a (possibly empty) message. Your 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
  - deleting an object owned by another client
- `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

Each client is assigned a unique `idNumber`, and the input file has command lines specific to each `idNumber`. 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, as needed. 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 a 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 each client program, the program is required to print information on all transmitted and received packets. In addition, it should indicate when it enters (and exits) a delay period.

The Server Loop

When the program works as a server, it uses I/O multiplexing (e.g., `select()` or `poll()`) to handle I/O from the keyboard and the FIFOs in a nonblocking manner. Each iteration of the main loop performs the following steps:

1. Poll the keyboard for a user command. The user can issue one of the following commands.
   - **list**: The program writes the stored information about the objects (the `idNumber` of the owning client, and the object name).
   - **quit**: The program exits normally.
2. Poll the incoming FIFOs from the clients. The server handles each incoming packet, as described above.

To monitor progress of the server, the program is required to print information on all transmitted and received packets.

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

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.

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' or 'make a2p2' 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 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 (to what degree the programs work as specified, e.g., tested and working, working with some known issues, etc.) mention difficulties encountered in the implementation
- **Testing and Results:** comment on how you tested your implementation, and discuss the obtained results
- **Acknowledgments:** acknowledge sources of assistance

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.

Marking

Roughly speaking, the breakdown of marks is as follows:

05% : ease of managing the project programs using the submitted Makefile

20% : Correctness, testing and results of Part 1

65% : Correctness, testing and results of Part 2

10% : quality of the information provided in the project report (if applicable: the results, justifications, discussions, etc.)
